VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

BAKALÁŘSKÁ PRÁCE

Brno, 2021 Martin Nohava

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

WEBOVÁ APLIKACE AUTENTIZAČNÍHO SYSTÉMU
WEB APPLICATION OF AUTHENTICATION SYSTEM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Martin Nohava**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. Lukáš Malina, Ph.D.**
SUPERVISOR

BRNO 2021

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Informační bezpečnost**

Ústav telekomunikací

Student: Martin Nohava ***ID:*** 211569
Ročník: 3 ***Akademický rok:*** 2020/21

NÁZEV TÉMATU:

Webová aplikace autentizačního systému

POKYNY PRO VYPRACOVÁNÍ:

Analyzujte soudobé možnosti návrhu a tvorby moderní webové GUI a webové aplikace, jako např. Progressive Web Apps, AWS, Node.js atd. včetně využití kontejnerů (docker) či otevřených cloudových řešení. Prozkoumejte dále možnosti integrace knihoven a API nižších programovatelných jazyků. Navrhněte a vytvořte demonstrační GUI webové aplikace, která rovněž umožní využívat kryptografické knihovny nižších programovacích jazyků.
V rámci bakalářské práce vytvořte funkční a bezpečnou implementaci grafického rozhraní autentizačního systému pomocí webové aplikace. Webová aplikace bude nabízet grafické rozhraní pro všechny entity autentizačního systému, tj. uživatelská část, ověřovatel, vydavatel a revokační autorita. Dalším cílem je zajistit komunikaci mezi entitami a bezpečné ukládání dat. Výsledný systém otestujte a využijte vhodné demonstrační prostředí, např. docker kontejnery.

DOPORUČENÁ LITERATURA:

[1] MENEZES, Alfred, Paul C VAN OORSCHOT a Scott A VANSTONE. Handbook of applied cryptography. Boca Raton: CRC Press, c1997. Discrete mathematics and its applications. ISBN 0-8493-8523-7.

[2] ATER, Tal. Building progressive web apps: bringing the power of native to the browser. O'Reilly Media, Inc., 2017.

Termín zadání: 1.2.2021 ***Termín odevzdání:*** 31.5.2021

Vedoucí práce: doc. Ing. Lukáš Malina, Ph.D.

doc. Ing. Jan Hajný, Ph.D.
předseda rady studijního programu

ABSTRAKT

Předmětem bakalářské práce je návrh a vývoj graficko-uživatelských rozhraní pro kryptografické protokoly pomocí webových technologií. Konkrétně přidává možnost ovládat autentizační systém PEAS skrze webová rozhraní. Práce nejprve vyhodnocuje soudobé možnosti vývoje moderních webových aplikací a věnuje pozornost problematice provázání webových technologií s autentizačním systém tvořeným v programovacím jazyce C. Pro tento účel se práce blíže věnuje také technologii WebAssembly.

V praktické části byla vytvořena čtyři grafická uživatelská rozhraní, jedno pro každou entitu autentizačního systému. Jsou postavena na webových technologiích HTML, CSS a JavaScript, za využití frameworku Vue.js. Komunikaci s autentizačním systémem zajišťují, pomocí API, webové servery, které jsou implementovány v jazyce JavaScript a spouštěné v prostředí Node.js.

KLÍČOVÁ SLOVA

autentizační systém, grafické rozhraní, JavaScript, Node.js, PEAS, PWA, RKVAC, Vue.js, WebAssembly, webová aplikace, webové frameworky

ABSTRACT

The subject of the bachelor thesis is the design and development of graphical-user interfaces for cryptographic protocols using web technologies. Specifically, it adds the possibility to control the PEAS authentication system through web interfaces. The thesis first evaluates the contemporary possibilities of modern web applications and pays attention to the issue of interconnecting web technologies with authentication system created in C programming language. For this purpose, the work also focuses on the Webassembly technology.

In the practical part, four graphical user interfaces were created, one for each entity of the authentication system. They are built on web technologies such as HTML, CSS and JavaScript, utilizing the Vue.js framework. Communication with the authentication system provides the developed web servers. They are using API and run in the Node.js environment.

KEYWORDS

Authentication system, Graphical interface, JavaScript, Node.js, PEAS, PWA, RKVAC, Vue.js, WebAssembly, Web application, Web framework

NOHAVA, Martin. *Webová aplikace autentizačního systému*. Brno, 2021, 71 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: doc. Ing. Lukáš Malina, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Webová aplikace autentizačního systému“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu bakalářské práce panu doc. Ing. Lukáši Malinovi, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci. Dále děkuji panu Ing. Petru Dzurendovi, Ph.D za konzultace a podnětné návrhy při vypracování praktické části mé bakalářské práce.

Tato práce vznikla jako součást klíčové aktivity KA6 - Individuální výuka a zapojení studentů bakalářských a magisterských studijních programů do výzkumu v rámci projektu OP VVV Vytvoření double-degree doktorského studijního programu Elektronika a informační technologie a vytvoření doktorského studijního programu Informační bezpečnost, reg. č. CZ.02.2.69/0.0/0.0/16_018/0002575.

EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

Projekt je spolufinancován Evropskou unií.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

V současné době je drtivá většina uživatelů moderních výpočetních technologií zvyklá na používání grafických rozhraní pro interakci s těmito zařízeními. Z historického vývoje se tento způsob ukázal, jako mnohem pohodlnější než zadávání textových příkazů, avšak tato metoda je též stále používána pro svou preciznost.

Jedním z trendů při vývoji GUI (*Graphical User Interface*) je tvořit tato rozhraní jako webové aplikace. Objevil se koncept *Progressive Web Applications*, který umožňuje používat webové aplikace podobně jako ty nativní. Hlavními výhodami tohoto přístupu jsou minimální nároky na koncová zařízení, kterým stačí pro spuštění takových aplikací pouze kompatibilní webový prohlížeč a přístup na internet. Tím lze zároveň najednou pokrýt i všechny platformy pomocí jediného softwarového řešení.

Nadcházející kapitoly 1 a 2 této práce se blíže věnují právě takovým webovým aplikacím a zkoumají soudobé trendy v jejich vývoji. Práce porovnává možnosti v současné době nejoblíbenějších a nejrozšířenějších frameworků[1] na trhu a vybírá nejvhodnější pro tvorbu grafické nadstavby pro již existující autentizační systém PEAS (*Privacy-Enhancing Authentication System*) představený v kapitole 4. Pozornost není upřena ani pravidlům pro tvorbu GUI a UX (*User Experience*), kterým je věnována vlastní kapitola 5.

Jak již bylo zmíněno, aplikace jsou určeny jako nadstavba nad autentizační systém, který je však vytvořen formou konzolového programu. Jednou z klíčových částí práce je tedy i průzkum v kapitole 3, jak a zdali vůbec je možné pomocí webových technologií a vybraných frameworků docílit kompatibilní komunikace s tímto programem.

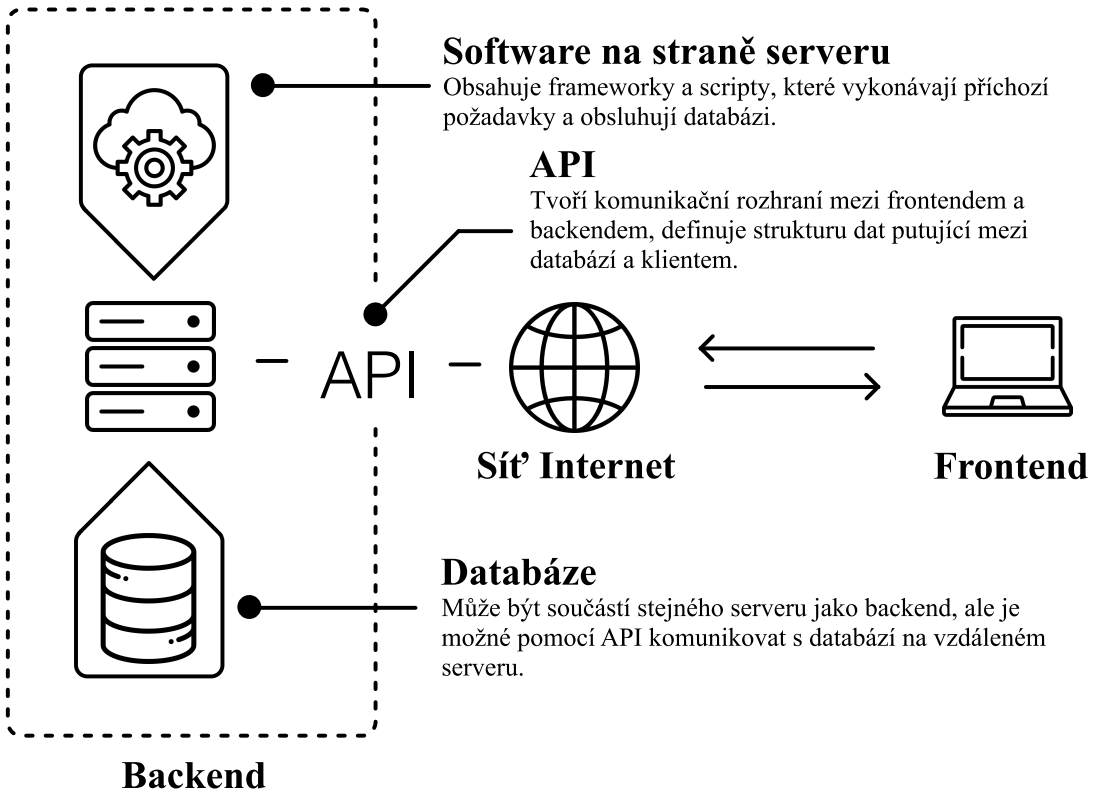
Cílem práce je poté zužitkování těchto informací v navazující praktické části 6 a vytvoření webových aplikací pro všechny entity autentizačního systému. Výsledkem by mělo být usnadnění uživatelské interakce s tímto systémem a snazší prezentace této technologie široké veřejnosti.

[1] Softwarová struktura, sloužící jako kostra při vývoji projektů.

1 Webové aplikace

Při vývoji webových aplikací vznikají programy, které je možné hostovat na webových serverech. Na rozdíl od vývoje klasického software pro personální počítače, který může často plnit svou funkci i bez připojení k síti, jsou webové aplikace závislé na internetu.

V současné době se tato technologie těší velkému zájmu a je diskutována možnost nahrazování nativních aplikací těmi webovými [1]. To skýtá určité výhody, například vývojářský tým poté spravuje pouze jednu aplikaci pro všechny platformy. Dnes se takto vyvíjejí především aplikace pro „konzumaci“ obsahu na internetu. Pro software na tvorbu obsahu (např. editaci videa) není, alespoň prozatím, tato technologie zcela vhodná. Avšak je možné, že v blízké budoucnosti dojde i zde k posunu díky nastupujícím novým technologiím jako je *WebAssembly*, viz kapitola 3.2.1. Architektura webových aplikací se dělí na dvě základní jednotky, frontend a backend. Viz následující obrázek 1.1.



Obr. 1.1: Schématické znázornění hlavních částí backendu.

Backend, jak jej definuje práce [2], se stará o logickou stránku webové aplikace. Na obrázku 1.1 je znázorněn čárkovaným ohraničením. Tvoří se pomocí programovacích jazyků, které lze spustit na počítačových serverech. Jedná se například

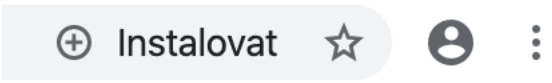
o scriptovací jazyky jako JavaScript, Python, ale i další jako Java. Stará se o práci s daty a definuje, jak budou uložena, zpřístupněna a poskytnuta uživatelům. Jeho hlavní součásti jsou scripty, frameworky, databáze a API (*Application Programming Interface*).

Na druhé straně vývoj frontendu se především zabývá estetickou částí aplikace. Stará se o srozumitelnou interpretaci dat uživatelům. Jednou z nejnáročnějších výzev je udržet konzistenci napříč různými zařízeními od počítačů po mobilní telefony tak, aby ani v jednom případě nijak neutrpěl uživatelský zážitek. Design webových stránek, použitelnost a uživatelská přívětivost jsou základními stavebními kameny při jeho vývoji [2]. Dále tyto otázky rozvijí kapitola 5. Výše na obrázku 1.1 je vidět umístění frontendu ve webové aplikaci.

Technologie Progressive Web Applications

Momentálně jsou tyto aplikace poměrně populárním tématem. Reprezentují stále poněkud nový koncept webu jako aplikace, který ale začaly prosazovat a podporovat ve svých prohlížečích velké společnosti jako Google a Microsoft.

Jde o jisté rozšíření funkcionality webových aplikací a přiblížení jejich možností k těm nativním. Nabízí *caching*[1] důležitých komponent a již zobrazených dat, jsou tedy rychlejší při vykonávání rutinních operací než standardní webové aplikace a v omezené míře fungují i offline. Dokáží využívat *push-notifikace* včetně dalšího API operačního systému a pracovat na pozadí, ale především jsou instalovatelné z webu přímo do zařízení pomocí webového prohlížeče, který jim slouží jako *runtime environment*. Viz obrázek 1.2. Jsou tedy více uživatelsky přívětivé, a to především na mobilních zařízení. Více o PWA (*Progressive Web Applications*) viz [3].



Obr. 1.2: Možnost instalace v Google Chrome.

Nejdůležitějšími komponenty definující progresivní webové aplikace jsou, dle článku [3], *Web App Manifest* a *Service Worker*. Manifestem se rozumí jednoduchý JSON (*JavaScript Object Notation*) soubor, který dává vývojářům možnost kontroly nad tím, jak se bude aplikace zobrazovat uživateli a to především po instalaci. Definuje její název, odkazuje na obrázky, které použít jako ikony aplikace po přidání na domovskou obrazovku, dovoluje aplikaci spustit se v režimu přes celou obrazovku bez adresního řádku a mnohem více [4].

[1] dočasné ukládání dat

Service Worker je script, který běží na pozadí odděleně od webové stránky. Odpovídá na vyvolané události v aplikaci včetně těch síťových. Má jen velmi omezenou životnost. Je probuzen přiřazením události a běží pouze tak dlouho, dokud nedojde k jejímu splnění. Využívá *Cache API* pro uložení dat a může tak fungovat i bez připojení k síti. Po opětovném připojení provede požadované operace a aktualizuje obsah [3]. Díky tomu lze pracovat s webovými aplikacemi, které implementují standardy PWA, podobně jako s nativními.

2 Porovnání vývojových prostředí

V případě, kdy je při vývoji webové aplikace uvažováno o zapojení frameworku, je v dnešní době zapotřebí jistá orientace v jejich nabídce. Různých prostředí pro vývoj existuje nepřeberné množství a každé má své silné a slabé stránky.

Je tedy důležité vědět, jaké parametry jsou pro požadovaný projekt zásadní a dopředu zvolit správné vývojové prostředí. Pokud je tento krok splněn, vývoj vlastní aplikace se stává o něco jednodušším.

Framework doplňuje aplikaci o standardní funkce jako například *routing* a *state-management*, které by jinak vývojář musel pokaždé sám implementovat. Definuje celkovou strukturu projektu a zavádí jistou míru standardizace, která ulehčuje orientaci napříč projekty.

Pozornost je věnována především těm nejrozšířenějším frameworkům, které jsou využívány velkým množstvím společností, týmy vývojářů a jsou dobře zdokumentovány. Posuzují se především jejich odlišnosti v přístupu k vývoji aplikací, jak snadné je začít se samotnou tvorbou webu, jaké nabízí vývojové prostředí a specifika každého z nich.

V závěru této kapitoly je uvedena tabulka 2.4 shrnující hlavní vlastnosti jednotlivých frameworků a rozhodnutí, který z nich byl vybrán. Následující podkapitoly je dělí dle použitého programovacího jazyka.

2.1 Vývojová prostředí využívající jazyka JavaScript

Jde bezesporu o nejpoužívanější jazyk při vývoji webových aplikací i jednodušších stránek, viz A.1. JavaScript se stal jakýmsi neoficiálním standardem pro web developement. Patří k rodině objektově orientovaných skriptovacích jazyků, jeho funkcí je zajišťovat chod dynamických prvků stránek, jako jsou různé formuláře, tlačítka a animace. Je často kombinován s HTML (*Hypertext Markup Language*), kdy může být psán přímo v těle stránky za použití tagu `<script>`, nebo je importován z externího zdroje [5]. Jedná se o takzvaný *event-driven*, neboli událostmi řízený jazyk.

Historicky, jak uvádí literatura [6], byl vykonáván webovými prohlížeči, které obsahují JavaScript engine. Ten umožňuje po stažení stránky začít na klientské straně provádět naprogramované chování webu. Dnes je díky technologiím jako Node.js ve velké míře využívána možnost provádění obsahu skriptu na straně serveru, to nabízí zcela nové možnosti včetně vyšší bezpečnosti. Lze tak například přistupovat k souborům na zařízení.

2.1.1 Platforma Node.js

Node.js se stal důležitou součástí a jednou z nejoblíbenějších voleb vývojářů při tvorbě aplikací postavených na jazyce JavaScript. Zmiňovaný je v této sekci z důvodu, že následující frameworky využívají tuto technologii a jsou jeho rozšířením. Slouží jako *runtime environment*, který se stará o běh programu po jeho přeložení. Skládá se z *open-source*[1] enginu *V8*, jenž je považován za jeden z nejlepších a je využívaný v prohlížeči Google Chrome. Tento engine je zabalený ve speciálně vytvořeném C++ programu, jehož spuštěním na zařízení lze překládat a debugovat JavaScriptový program, nebo dokonce vytvořit webový server [7].

Nedílnou součástí Node.js je *npm*, tedy *node package manager* v překladu správce balíčků node. Jeho existence velmi ulehčuje správu rozšíření a dalších závislostí, které jsou potřeba pro správné fungování vyvíjeného projektu [7]. V současné době *npm*, dle oficiálních stránek[2], spravuje přes jeden milión balíčků, ty pomáhají snadno implementovat různou funkcionalitu stránek, jakou může být přihlášení se ke službě pomocí účtu Google. Lze takto rychleji stavět na již připravených funkcích a zaměřit se na jádro vyvíjené aplikace.

Asynchronní architektura

Platforma Node.js je vhodná pro tvorbu vysoce škálovatelných a data intenzivních aplikacích reálného času. Používá pro to *non-blocking*[3] architekturu, tím se odlišuje od ostatních prostředí jako je ASP.NET, které využívá synchronní architekturu, a pro přechod na asynchronní je potřeba další práce. Jak uvádí článek [8], neblokující přístup za pomocí jediného vlákna zvládá obsluhovat několik požadavků naráz. V případě že požadavek vyžaduje časově náročnou operaci, jakou může být čtení ze souboru a tento zdroj není momentálně dostupný, přejde vlákno k obsloužení dalšího požadavku. Ve chvíli, kdy dojde k uvolnění zdrojů a přečtení dat ze souboru, vloží se tato skutečnost do fronty událostí. Vlákno neustále na pozadí monitoruje vyřízené události ve frontě a ihned je vrací klientům.

Tento přístup je výhodný pro použití v situacích, kdy je očekáván vysoký počet souběžných požadavků. V případě synchronní architektury by po čase došlo k vyčerpání všech vláken systému. Následně by docházelo k prodlužování doby, za kterou by byli klienti obslouženi. Asynchronní přístup tedy využije prostředky hardwaru mnohem efektivněji. Není však vhodný v případech, kdy je zapotřebí intenzivních výpočtů na CPU (*Central Processing Unit*), jako je práce s videem, či obrazem.

[1] Software s volně dostupným zdrojovým kódem.
[2] Více viz https://npmjs.com/
[3] neblokující

Z důvodu použití pouze jednoho vlákna v případě složitých výpočtů, je toto vlákno dlouho blokováno jediným dotazem a ostatní klienti musejí čekat.

2.1.2 Framework Angular

Za vznikem frameworku Angular stojí internetový gigant Google. Hlavní filozofií frameworku je nabídnout vývojářům nástroj, který bude tzv. vše v jednom. Angular obsahuje obrovské množství funkcí, které mají zabezpečit vývoj komplexních aplikací, a ne pouze grafického rozhraní.

Ve výchozím stavu má vestavěné moduly pro routing, HTTP dotazování, state management a mnoho dalšího. Proto je často nazývaný spíše platformou nežli frameworkem. Díky snaze společnosti Google v posledních letech „tlačit“ vývoj *Progressive Web Applications* dopředu, se tato snaha propisuje i do Angular, který nabízí integraci a podporu PWA takzvaně *out of the box*[4]. Další rozbor viz [9].

Vývojové prostředí

Při psaní kódu se dělí na dvě zásadní sekce. První obsahuje HTML kód odpovídající za vykreslování prvků na stránce a druhá využívající TypeScript pro naprogramování chování jednotlivých komponent. TS (*TypeScript*) je superset jazyka JavaScript a přidává do něj další volitelné prvky. Pro programátory orientující se v JavaScriptu by měl být proces adaptace a seznámení s tímto jazykem velmi rychlý. Proto je zařazen v této sekci. Zároveň existuje možnost transkompilace, tedy kompilace TypeScript kódu do JavaScriptu za pomocí kompilátorů jako Babel či Checker.

Při posuzování, jak rychle lze v tomto prostředí začít tvořit aplikace, je potřeba zohlednit komplexitu celého Angular. Pro začínající vývojáře je často velmi nepřehledný. V prvním kroku je nutné správné nastavení projektu, jelikož nelze použít jeden univerzální. Díky nástroji CLI (*Command Line Interface*) je tento proces o něco snazším. Angular zároveň není možné integrovat pouze do části již existující webové aplikace.

Shrnutí frameworku

Tento framework je tedy výhodné použít pro vytváření zcela nových aplikací, je připraven zvládnout i velmi komplexní projekty. Jeho hlavní součásti jsou spravovány oficiálním týmem, díky čemuž jsou vždy aktuální a vzájemně kompatibilní. Ne u všech frameworků je toto pravidlem. Na druhou stranu použití TypeScriptu a obsáhlost funkcí tvoří hlavní bariéru pro začínající vývojáře a malé týmy.

[4] ve výchozím stavu

2.1.3 Framework React

Další z řady populárních frameworků, spravovaný společností Facebook, který je svou filozofií jakýmsi protikladem výše zmiňovaného Angular. Snaží se o maximálně minimalistický přístup. Svým zaměřením se hodí především na tvorbu grafických uživatelských rozhraní. Lze jej přirovnat spíše ke knihovně.

Lze jej využít i na velké projekty, potřeba však v tomto případě počítat s nutností konfigurace a instalace dodatečných rozšíření a vyčlenění prostředků na jejich správu. Ta může být obtížnější než u ostatních řešení, jelikož i ty nejdůležitější komponenty jako *router* jsou udržovány komunitou a při vydávání nových verzí může docházet k nekompatibilitám. Podrobnější informace lze nalézt v práci [9].

Vývojové prostředí

Ve fázi samotného programování se React odlišuje svým přístupem, kdy je vše tvořeno za pomocí JavaScriptu. Nejedná se ale o jeho tradiční implementaci, ale o takzvaný JSX, který je syntaktickým rozšířením JS (*JavaScript*). JSX umožňuje díky tomu zabalit segmenty HTML do JavaScriptové syntaxe, jak demonstruje následující výpis 2.1.

```
const element = <h1>Hello, {name}!</h1>;
```

Výpis 2.1: Ukázka syntaxe jazyka JSX

Nejde tedy o pouhý string uložený do konstanty, jelikož v něm lze používat proměnné označené složenými závorkami. S těmi lze následně pracovat jako při použití klasického JS a dynamicky měnit uživateli vykreslovaná data.

Shrnutí frameworku

Obtížnost pro založení projektu a odstartování vývoje aplikace v tomto prostředí je tedy ovlivňována několika základními faktory. Minimalistickým přístupem celého React, kde je poměrně jednoduché i pro nové programátory se zorientovat, ale s narůstající náročností projektu někdy narůstá i náročnost pro správu celého prostředí. Využitím JSX a psaním veškerého kódu do jediného bloku se poněkud snižuje přehlednost projektu. Je tedy potřeba nejprve dobře ovládat tuto specifickou syntaxi.

2.1.4 Framework Vue.js

Jediný z vybraných frameworků v této podkapitole, za nímž nestojí velká společnost. Jde o *open-source* projekt, na kterém spolupracuje nezávislý tým vývojářů. V porovnání s oběma výše uvedenými prostředími, se svojí filozofií nachází pomyslně

někde uprostřed. Zaměřuje se na hlavní pilíře při stavbě webových aplikací. Přichází s vlastním *routerem*[5], nabízí vestavěný *state-management* a oba tyto prvky jsou spravovány týmem Vue. Jsou tedy vždy kompatibilní a snadno použitelné. Více o Vue.js viz [9].

Vývojové prostředí

Proces vytvoření a spuštění projektu je velice jednoduchý díky nástroji Vue CLI, který vytvoří základní strukturu a soubory aplikace. Velikou výhodou je po zadání příkazu `vue ui` možnost spuštění grafického rozhraní pro správu projektů. Ta umožňuje velice názornou práci s nastavením, správou závislostí a rozšířeními vyvíjené aplikace. Informuje programátora i o výkonu kódu pomocí dashboardu a statistik.

Samotný kód je dělen do souborů, které jej logicky dělí na jednotlivé prvky[6] stránky. V každém ze souborů se nacházejí tři oddělené sekce. První označena tagem `<template>` obsahuje HTML komponentu, která je následně vykreslována uživateli. Tradiční HTML je zde rozšířeno o speciální znaky pro provázání s JavaScriptem. Jedním z nich je například `v-on:click="greet"`, který volá funkci s názvem greet při kliknutí na element.

Tyto funkce jsou definovány v následující sekci ohraničené tagem `<script>`. Ta je psána za využití standartní verze JavaScriptu. Pro vývojáře ovládající tento jazyk je tedy velice snadná na pochopení. Proměnné, objekty a funkce definované v této sekci se dále dělí na menší celky. Například funkce `data()` uchovává stav aplikace a její proměnné, objekt `computed:` obsahuje metody pro výpočet složitějších zobrazovaných údajů, `watch:` obsahuje funkce reagující na změnu určitých dat atd.

Poslední částí je sekce `<style>`. Zde je definován kaskádový styl, který je uplatňován na výše uvedeném HTML. Speciální funkcí je použití přepínače `scoped`, jenž omezí aplikaci stylu pouze na tento prvek stránky.

Shrnutí frameworku

Svým přístupem tedy Vue udržuje spolu související části kódu spojené přehledně do jediného souboru a zároveň zachovává jeho oddělení do samostatných sekcí. Díky tomu stačí znalost jednotlivých použitých jazyků, bez potřeby seznamovat se s novou syntaxí. V kombinaci s grafickou správou projektu se jedná o jednu z uživatelsky nejpřívětivějších voleb.

[5] Část frameworku sloužící pro mapování URL adres k jednotlivým stránkám webové aplikace.
[6] Za prvek stránky lze považovat například navigační panel.

2.2 Vývoj webových aplikací pomocí programovacího jazyka Python

Python patří mezi soudobé nejvíce populární jazyky, jak naznačuje graf A.1. Díky jeho popularitě vznikla rozšíření pro jeho využití při tvorbě webových aplikací [2]. Nejrelevantnější z nich jsou Flask a Django. Jedná se o *general-purpose*[7] jazyk, tedy na rozdíl od JavaScriptu nebyl tvořen primárně za účelem vývoje stránek, avšak právě díky tomu je případná implementace komplikovanější výpočetní logiky v backendu příjemnější. Jde o interpretovaný jazyk, není tedy potřeba jeho kompilace do strojového kódu. Místo toho za jeho běh odpovídá interpret, který je dostupný pro mnoho platforem, zahrnující i webové servery.

Jedním z nástrojů usnadňující práci s projekty Pythonu je *pip*, tedy manažer balíčků. Každý balíček obsahuje všechny soubory potřebné pro správnou funkci určitého modulu. Ty jsou rozšířeními, které lze přidat do projektu v podobě knihoven.

2.2.1 Framework Flask

Je jednodušším ze dvou jmenovaných frameworků. Nabízí pouze základní funkcionalitu pro vytvoření webových aplikací v krátkém čase. Implementuje pouze ty nejzákladnější metody a další práci nechává na vývojářích. Ti následně mohou pracovat mnohem flexibilněji při konstrukci aplikace. Další popis obsahuje studie [2].

Vývojové prostředí

Zakládání projektu ve Flasku je kontrastem ke všem předešlým prostředím, kde pomocí CLI či jiných nástrojů byla generována základní struktura a soubory projektu. Zde lze jednoduše vytvořit hlavní soubor v jazyce Python a začít psát kód. Filozofie adresářů je však podobná, využívá podsložky templates, kde jsou uloženy HTML soubory a podadresáře static, který obsahuje CSS a případně dodatečný JavaScript.

V hlavním souboru jsou pomocí dekorátorů `@app.Route()`, o které tento framework rozšiřuje Python, definovány URL (*Uniform Resource Locator*) adresy a k nim vázané funkce, které jsou při jejich zadání do prohlížeče spuštěny. Ty s dalšími parametry nejčastěji odkazují na HTML soubory obsahující frontend [2].

Flask nijak nerozšiřuje a nezasahuje do samotné syntaxe jazyka HTML, ale umožňuje psaní příkazů a vkládání proměnných v jazyce Python přímo v těchto souborech, jak ilustruje výpis 2.2.

[7] pro všeobecné využití

```
<body>
    {% for post in posts %}
        <h1>{{ post.title }}</h1>
        <p>{{ post.content }}</p>
    {% endfor %}
</body>
```

Výpis 2.2: Ukázka kombinace HTML a jazyka Python.

Ukázka 2.2 demonstruje vypsání elementů z listu `posts` pomocí cyklu `for` a vložení dat, které obsahují, tedy `title` a `content`, do HTML tagů. Takto dynamicky vkládaný obsah je následně zobrazen uživateli v graficky formátované podobě. Notace `{%...%}` ohraničuje vkládané příkazy, při nahrazení procent za složené závorky vypisujeme hodnoty proměnných `{{...}}`.

Shrnutí frameworku

Nasazení tohoto frameworku při tvorbě projektu je velice jednoduché a efektivní. Lze s ním vytvořit prototyp jednoduché aplikace v řádu hodin. Nevyžaduje žádnou speciální syntaxi. Nevýhodou může být potřeba vlastní implementace pokročilých funkcí, pokud jsou potřeba.

2.2.2 Framework Django

Někdy označováno termínem *battery-included*[8] framework je řešením pro rapidní vývoj webových aplikací. Jinými slovy obsahuje již od prvního spuštění veškeré potřebné nástroje pro tvorbu grafického rozhraní a jeho programového základu. Zároveň je *open-source* projektem a lze jej tedy dál jednoduše modifikovat jako celek. Blíže se mu věnuje studie [2].

Vývojové prostředí

Zahájení práce v Djangu je podobné výše zmiňovaným prostředím Angular, React a Vue frameworků. Pomocí manažera balíčků *pip* lze snadno nainstalovat Django CLI, které pomocí příkazu `django-admin startproject` automaticky vygeneruje novou adresářovou strukturu projektu obsahující všechny základní soubory. Těmi jsou především soubory s nastavením aplikace, *routeru* a soubor `manage.py`, který umožňuje řízení projektu a spouštění vývojářského serveru pomocí textových příkazů.

Celý projekt se dělí do logických jednotek (aplikací), které tvoří jednotlivé části webu. Nejedná se však o jednotlivé komponenty ve smyslu, v jakém s nimi pracuje

[8] Výraz označující připravenost produktu plnit svou funkci, bez nutnosti dalšího přičinění uživatele.

např. Vue. Zde se jedná o větší celky jako jsou celé stránky. Ty však lze opětovně používat na více místech nebo v úplně jiném projektu. Šetří tak drahocenný čas při vytváření produktu. Každá z těchto jednotek je opět automaticky generována pomocí CLI v rámci projektového adresáře. Django tak sice nabízí velmi pohodlně automatizovanou tvorbu struktury budoucí webové aplikace, ale pro nové vývojáře může být skutečnost, kdy ještě nezačali reálně programovat a už mají vygenerované velké množství souborů, zastrašující.

Samotný kód aplikace je tvořen jazyky Python, HTML a CSS (*Cascading Style Sheets*). Pro programátory obeznámené s těmito jazyky by mělo být poměrně jednoduché pochopit celou filozofii frameworku a začít být rychle produktivní. Pro zobrazování dynamických prvků na stránkce používá identickou syntaxi jako Flask. Příkazy v jazyce Python se ohraničují `{%...%}` a proměnné `{{...}}`. Stejně jako ve Flasku i zde musí být příkazy ohraničené koncovými tagy. Například `if` musí být ukončený řádkem `{% endif %}`. Jinak se jedná o nijak nemodifikované jazyky, které v případě Pythonu pouze využívají knihoven Django.

Jednotlivé komponenty na stránce (tlačítka, formuláře apod.) samozřejmě lze rozdělit do jednotlivých šablon, jako znovu použitelné komponenty. Těmito komponenty jsou tvořeny, výše zmiňované, větší logické celky (aplikace).

Shrnutí frameworku

Využití frameworku Django poskytuje při vývoji znatelně vyšší oporu o již předpřipravenou funkcionalitu a knihovny, než nabízí přímý konkurent Flask. Řeší mnoho věcí za programátora, jako například i implementaci databáze, kterých podporuje velké množství. Nevyžaduje pochopení nových konceptů při psaní samotného kódu, ale hierarchie a struktura souborového systému projektu může být poněkud náročnější.

2.3 Vývoj webových aplikací v prostředí využívající jazyk C#

Jazyk C# je moderním objektově orientovaným jazykem, který podporuje hlídaní typů proměnných, jejich inicializaci a obsahuje také garbage collector pro automatickou správu paměti. Původně byl C# vyvíjen jako reakce na populární jazyk Java a sdílí s ním podobnou základní filozofii. Je vyvíjený společností Microsoft a speciálně cílený na platformu .NET Framework [10].

.NET je *runtime environment* obsahující velké množství dodatečných knihoven, které dramaticky ulehčují vývoj moderních aplikací cílených na různé platformy.

Poskytuje kompilátor nejen pro C#, ale i pro ostatní jazyky, které jsou součástí tohoto frameworku, jako například funkcionálně zaměřený F#.

Jeho dvě největší součásti jsou tedy *runtime environment*, který se nazývá *Common Language Runtime* (CLR) a *Framework Class Library* (FCL), která je postavena na CLR a poskytuje služby moderním aplikacím. Blíže jsou všechny vlastnosti tohoto frameworku vysvětleny v literatuře [10].

Relativní novinkou je .NET Core *runtime*, který je *open-source* a umožňuje spouštění aplikací na obrovské škále zařízení od počítačů s Windows, Linux, MacOS po IoT zařízení.

2.3.1 Framework ASP.NET

Je částí ekosystému .NET, která je optimalizovaná pro vývoj dynamických webových aplikací. Využívá nejčastěji C# pro tvorbu logiky a kombinaci HTML a CSS pro frontend. Řídí se MVC (*Model-View-Controller*) architekturou. Ta je složena ze tří hlavních částí, *Model*, *View* a *Controller*.

Jak tyto komponenty popisuje článek [11], *model* reprezentuje logickou část aplikace, která je nezávislá na frontendu. Lze jí tedy vyjmout a následně v rámci frameworku použít třeba při tvorbě mobilní aplikace. *View* je část zobrazovaná uživateli, tedy HTML *markup*. *Controller* je odpovědný za práci s HTTP požadavky na aplikaci a kombinuje modely z databáze, které vkládá do *View*, jenž se zobrazí uživateli. Za volbu správného *Controlleru* se stará *router*.

Stejně jako u ostatních frameworků i zde existuje *package manager*, který dovoluje rozšiřovat projekt o další funkcionalitu a balíčky. Zde se nazývá *NuGet* a v současné chvíli obsahuje více než 230 000 balíčků[9].

Vývojové prostředí

Prostředí pro vývoj aplikace se liší dle zvolené verze .NET. V případě zvolení .NET Core jej lze nainstalovat na libovolný operační systém, ale poskytuje pouze CLI pro ovládání frameworku. Verze .NET Framework je integrována do vývojového prostředí programu Visual Studio a nabízí grafické rozhraní pro tvorbu, výběr šablon a následnou správu projektu. Velkou limitací však může být kompatibilita pouze s operačním systémem Windows.

Jako hlavní programovací jazyk, tedy tento framework využívá C# v kombinaci s HTML a kaskádovými styly. S HTML však pracuje po svém a do značné míry jej modifikuje. Soubory obsahující tuto kombinaci HTML a C# mají příponu .cshtml a v MVC modelu odpovídají *view* komponentu. Pomocí notací jako `@{...}` lze do HTML vkládat kód C#.

[9] Více viz https://www.nuget.org/

Shrnutí frameworku

Práce s ASP.NET frameworkem je poměrně komplexní záležitostí a vyžaduje dobrou znalost základů jazyka C#. Dále je nutné porozumnění konceptům tvorby webových aplikací v modelu MVC. Je potřeba si osvojit strukturu projektu, který má důležité komponenty rozesety do mnoha adresářů. Samozřejmě toto řešení nabízí velmi dobrý základ pro tvorbu velkých aplikací, ale svým přístupem působí komplikovaně pro nové vývojáře.

2.4 Vyhodnocení frameworků

Následující tabulka shrnuje obecné vlastnosti porovnávaných frameworků. V případě kategorie *routing* je brán stav při výchozím nastavení projektu. V některých případech je tedy možné, nechat si doinstalovat tuto funkcionalitu samotným frameworkem i dodatečně.

Integrace knihoven nižších programovacích jazyků, konkrétně jazyka C, je blíže popsána v následující kapitole 3. Disponují jí však v nějaké podobě všechny frameworky.

Tab. 2.1: Základní přehled důležitých vlastností frameworků.

Framework	Jazyk	PWA	Routing	Integrace knihoven C
Angular	TS (JS)	ANO (plugin)	ANO	ANO (*.wasm)
React	JSX (JS)	ANO (plugin)	NE	ANO (*.wasm)
Vue.js	JS	ANO (plugin)	NE	ANO (*.wasm)
Flask	Python	ANO (manuálně)	ANO	ANO (*.so)
Django	Python	ANO (manuálně)	ANO	ANO (*.so)
ASP.NET	C#	ANO (plugin)	ANO	ANO (*.dll)

Po zvážení všech uvedených možností byl vybrán framework **Vue.js**. Pracuje s minimálně modifikovaným jazykem JS, který je nejpopulárnější volbou pro web developement. Díky tomu existuje velké množství podpůrných materiálů, pluginů a dokumentace, ze které může tato práce profitovat. Další výhodou použití jazyků bez pozměněné syntaxe, je rychlé seznámení se s frameworkem pro vývojáře disponující zkušenostmi s vývojem webu.

Nabízí velmi přehledné členění struktury projektu. Tu doplňuje jedinečná možnost využívat namísto klasického CLI pro management projektu grafické webové rozhraní. Zde je k dispozici přehledná správa veškerých závislostí, statistik a nastavení.

I když *state-management* a *routing* není automaticky implementovaný při vytvoření nového projektu, díky právě zmiňovanému GUI je přidání těchto funkcí pouze otázkou zmáčknutí tlačítka. Stejným způsobem je možné přidat kompletní podporu standardu *Progressive Web Applications*.

Při použití technologie *WebAssembly* lze i s tímto frameworkem integrovat napsané knihovny v jazyce C a volat v nich obsažené kryptografické funkce.

Celkově tedy nabízí tato varianta podporu pro rapidní vývoj webové aplikace, bez potřeby zvláštních prerekvizit jakou je znalost specifické syntaxe. Jde tedy o jednu z nejjednodušších platforem pro vývojáře, přičemž tím ale nijak netrpí kvalita projektu a jeho funkcí. Podporuje všechny požadované parametry jako integraci knihoven nižších programovací jazyků. A disponuje jednou z nejlepších dokumentací.

3 Možnosti integrace knihoven a API

Tato kapitola práce zkoumá možnosti napojení webové aplikace autentizačního systému na externí programy pomocí API a využití již existujících kryptografických knihoven napsaných v jazyce C.

3.1 Komunikace pomocí API

V obecném označení je API soubor funkcí, které dovolují aplikaci komunikovat s externími softwarovými komponenty, operačními systémy a dalším softwarovým vybavením [12].

Ve webovém prostředí se nejčastěji využívá REST API, které dovoluje vývojářům pomocí volání požádat o určitá data anebo je poslat. Ta jsou většinou předávána ve formátu JSON. Data v tomto formátu jsou poměrně snadno čitelná pro lidi i stroje. Příklad objektu uživatele v soboru typu JSON je uveden ve výpisu 3.1. Je zde viditelné používání klíče a hodnoty. Kde na levé straně je klíč, který zůstává napříč objekty stejný (`"name"`) a jeho proměnná hodnota uvedená napravo (`"John Doe"`). Další výhodou je styl zápisu podobný JavaScriptu. To je jazyk, kterým se nejčastěji takto obdržená data ve webových aplikacích zpracovávají. REST API používá čtyři standardizovaná volání.

- **GET** – Vyžádá specifická data ze vzdáleného serveru.
- **PUT** – Požádá o aktualizaci konkrétních dat.
- **POST** – Vloží nová data do vzdáleného umístění.
- **DELETE** – Požádá o vymazání určených dat.

```
1  {
2    "name": "John␣Doe",
3    "admin": false,
4    "age": 21,
5    "tokens": [
6      {"id": 1, "type": "location", "value": "Czechia"},
7      {"id": 2, "type": "gender", "value": "male"}
8    ]
9  }
```

Výpis 3.1: Ukázka zápisu dat v souboru JSON.

Komunikace v rámci API tedy probíhá pomocí dotazování. Součástí každého dotazu je následujících několik komponent. Jejich bližší popis viz [12].

Koncový bod

Je vstupním bodem pro komunikaci a je nejčastěji reprezentován pomocí URL adresy, na kterou jsou následně směřovány dotazy API. Tato adresa je složena ze dvou klíčových částí. Jednou je adresa samotného serveru nebo služby, například `https://api.peas.com/`. Druhou je cesta, která specifikuje, k jakým datům bude přistupováno. Příkladem může být vyžádání seznamu uživatelů, tedy zaslání dotazu na koncový bod s adresou `https://api.peas.com/reg/users/`.

Hlavička

Obsahuje důležité informace o komunikaci pro klienta i server. Typicky hlavička obsahuje *autentizační token*, který je klientovi vystaven při vytvoření API účtu u serveru. Ten slouží pro jeho následnou identifikaci. Další důležitou součástí je například *content type* specifikující druh dat, která budou předávána. Pro soubory typu JSON jde o hodnotu `application/json`.

Metoda

Udává, jakou akci požaduje klient se specifikovanými daty provést. Tyto metody jsou uvedeny v seznamu výše.

Data

Jsou také často označována jako tělo (`body`) požadavku nebo odpovědi, pokud jsou data pouze požadována. Ukázkou takových dat obsažených v těle může být výpis 3.1.

Funkcionalita API je všeobecně brána jako jedna ze základních funkcí webových aplikací a lze ji tedy v nějaké formě implementovat do každého z výše uvedených webových frameworků.

3.2 Integrace knihoven jazyka C

Jedním z požadavků na vyvíjenou webovou aplikaci je prozkoumat a zajistit její případnou schopnost integrovat a využívat již připravené kryptografické knihovny v jazyce C. Zmiňovaný jazyk však nikdy nebyl zamýšlen pro použití na webu. Je tedy nutné utilizovat nějaký z nástrojů, který tento kód a metody v něm obsažené zpřístupní vyšším programovacím jazykům, na kterých jsou postaveny webové aplikace.

3.2.1 WebAssembly

Hovoří se o něm jako o další nejdůležitější inovaci v oblasti vývoje webových aplikací od představení JavaScriptu společností Netscape v roce 1995. Nesnaží se však nahradit JavaScript, nýbrž ho doplnit tam, kde se v minulosti ukázal jako ne zcela vhodný. *WebAssembly* též známý jako *Wasm*, se poprvé objevil v roce 2017 a je výsledkem spolupráce mezi společnostmi W3C, Google, Apple, Mozilla a Microsoft [13].

WebAssembly standard definuje spustitelný binární formát, korespondující textový přepis, který je podobný jazyku assembly a rozhraní pro interakci s hostujícím systémem. Využívá se jako *compilation target*, tedy kód do něhož jsou při kompilaci přeloženy programovací jazyky jako C, C++, Rust [13]. Vyjmenované jazyky jsou nejčastěji používané, jelikož nevyužívají automatický garbage collector a další komplikované konstrukty k implementaci a jsou pomyslně pouze o stupeň výše než strojový kód. Postupně se ale rozrůstá podpora a dnes již podporuje okolo čtyřiceti *high-level* jazyků jako Go, Java a PHP.

Wasm není nový jazyk, ale kompaktní set binárních instrukcí, který je agnostický k platformě, na které běží, primárně je však určený pro web. Využívá jej virtuální stroj implementovaný v prohlížeči. Jeho hlavní předností je *near-native* rychlost vykonávání kompilovaných instrukcí. Dovoluje tedy nově vývojářům psát kód v libovolném z podporovaných jazyků a využít tyto programy na webu. Více o *Wasm* viz [13].

Z pohledu bezpečnosti je to též velký krok kupředu. Jak si všímá studie [14], webové aplikace mají velmi málo možností, když přijde na rychlé a věrohodné kryptografické knihovny. Lze využít W3C WebCrypto API, které však nabízí limitovaný výběr algoritmů. Chybí podpora moderních standardů jako Curve25519, Chacha20, Poly1305, SHA-3 nebo Argon2i. Pokud tedy vývojáři potřebují využít nějaký z nepodporovaných algoritmů, musejí se spoléhat na často nedokonalé vlastní implementace v JavaScriptu. Díky *WebAssembly* lze využít knihoven napsaných v jiných jazycích, nebo jako v případě této práce speciálních kryptografických knihoven.

Emscripten

Jde o *open-source* kompilátor do *WebAssembly*. Podporuje kompilaci jazyků C a C++, nebo celých *runtime* prostředí v těchto jazycích napsaných. Po instalaci jej lze volat v příkazovém řádku pomocí `emcc`.

Zdrojové soubory lze kompilovat a volat funkce v nich obsažené několika způsoby, které popisuje článek [15]. Pokud budeme uvažovat ukázkový kód v jazyce C reprezentovaný výpisem 3.2, který bude uložen v souboru `function.c`. Zavolání příkazu

`emcc function.c -o function.html -s EXPORTED_FUNCTIONS="['_sum']"` vygeneruje tři soubory. Těmi budou `function.html`, `function.js` a binární soubor obsahující kód C `function.wasm`.

HTML soubor slouží pouze pro testování správné funkčnosti kompilovaných souborů a málokdy je reálně zapotřebí. Proto se častěji používá, při specifikaci výstupního souboru, přepínač `-o function.js`, který vynechá tento soubor a výstupem jsou pouze `function.js` a `function.wasm`.

Pomocí `function.js` lze následně volat funkce obsažené v binárním souboru. Ten obsahuje ale pouze ty vybrané přepínačem `EXPORTED_FUNCTIONS`.

Další možností je nespecifikování jednotlivých funkcí pro export v samotném příkazu, ale přímo v kódu pomocí anotace `EMSCRIPTEN_KEEPALIVE` viz výpis 3.2. Poté stačí již zavolat příkaz `emcc function.c -o function.js`.

```
1  EMSCRIPTEN_KEEPALIVE
2  int sum(int a, int b) {
3    return a + b;
4  }
```

Výpis 3.2: Příklad označení funkce v jazyce C pro export do modulu.

Volání funkcí je následně v JavaScriptu realizováno pomocí jedné ze dvou metod. První je `cwrap()`, díky které lze „zabalit“ danou funkci a následně ji opakovaně volat, nebo použitím druhé `ccall()` docílíme přímého zavolání funkce a vrácení její návratové hodnoty.

V prvním případě je potřeba funkci `cwrap()` předat název funkce k „zabalení“, návratovou hodnotu této funkce a pole s typy vstupních parametrů této funkce. Příklad zápisu je uveden ve výpisu 3.3.

```
1  const sum = Module.cwrap('sum',
2                           'number',
3                           ['number', 'number']);
4
5  console.log(sum(1,2)); \\Vypíše 3
6  console.log(sum(3,4)); \\Vypíše 7
```

Výpis 3.3: Zabalení funkce pomocí cwrap().

V případě zvolení funkce `ccall()` je zapotřebí navíc dodat ještě vstupní hodnoty funkce. Příklad je uveden ve výpisu 3.4.

```
1    let result = Module.ccall('sum',
2                              'number',
3                              ['number', 'number'],
4                              [1, 2]);
```

Výpis 3.4: Zabalení funkce pomocí ccall().

3.2.2 ctypes

Jde o nástroj obsažený ve standardní knihovně Python, který umožňuje vytvářet propojení mezi kódem v jazyce C a Python. Nabízí nízko úrovňovou množinu nástrojů pro načtení sdílených knihoven a *marshalling* [1] dat mezi Pythonem a C [16].

Nejdříve je potřeba pomocí kompilátoru uložit požadovaný kód v jazyce C, jako sdílenou knihovnu do souboru `.so`. Toho lze docílit příkazem `cc -fPIC -shared function.c -o lib.so`, kde přepínač `-fPIC` informuje překladač, že se bude jednat o *Position Independent Code*. Z důvodu, že jde o sdílenou knihovnu a nelze určit kolik programů a jak budou knihovnu využívat, zaručí tento přepínač použití relativních adres paměti a ne absolutních, jak vysvětluje článek [18].

V samotném kódu je tedy zapotřebí importovat tyto nástroje pomocí `import ctypes`. Dále načíst sdílenou knihovnu obsahující požadovaný kód v jazyce C. Důležité je specifikovat typ vstupních parametrů funkce. Následně již lze zavolat funkci a zaznamenat její návratovou hodnotu.

Pokud bude uvažovaný výchozí kód v C stejný jako ve výpisu 3.2, jeho načtení ve zdrojovém souboru Python bude následující, viz výpis 3.5.

```
1    import ctypes
2    x = 1
3    y = 2
4    imported = ctype.CDLL("lib.so")
5    imported.sum.argtypes = [ctypes.c_int]
6    result = imported.sum(x, y)
```

Výpis 3.5: Využití funkce sdílené knihovny v Python.

3.2.3 Dynamic link library

Neboli DLL jsou knihovny, které nabízejí znovu použitelný kód, na němž mohou stavět další programy. Na rozdíl od spustitelných souborů, funkce v nich obsažené potřebují externí program, který je zavolá a nelze je spustit přímo [19].

[1] Marshalling je název procesu, kde dochází k transformaci reprezentace dat v paměti. Je potřeba při použití výstupu programu, jako vstupu pro program napsaný v jiném jazyce [17].

Touto metodou je možné načíst knihovny napsané v C do projektu jazyka C#. Pro načtení DLL se využívá atribut `DllImport`. Ten značí že se bude jednat o kód, který je vykonáván mimo CLR. Při volání funkcí pomocí tohoto atributu je zapotřebí dopředu znát jména jednotlivých funkcí a typ jejich návratové hodnoty. Více viz článek [20]. Jednoduchým příkladem načtení externí funkce z DLL je výpis 3.6.

Nejprve je však zapotřebí kompilace zdrojových kódů knihovny do formátu DLL. Toho lze docílit například za pomocí programu `gcc`. Před každou funkci která je určená pro export do sdílené knihovny, musí být vloženo označení `__declspec( dllexport )`. Následně provedením samotné kompilace příkazem `gcc -shared function.c -o lib.dll` vznikne soubor dynamické knihovny `lib.dll`.

```
1       using System;
2       using System.Runtime.InteropServices;
3
4       class Example
5       {
6           [DllImport("lib.dll")]
7           public static extern int sum(Int a, Int b);
8
9           static void Main()
10          {
11            sum(1, 2);
12          }
13      }
```

Výpis 3.6: Načtení DLL v aplikaci C#.

4 Systém atributové autentizace

Tento systém je základním kamenem okolo kterého je vytvořena celá práce. Tato kapitola představí jeho základní funkce a vlastnosti. Zároveň definuje, jaké budou stěžejní nároky kladené na navrhované webové aplikace.

4.1 Funkce systému

V současné době masivní digitalizace, je kladen čím dál tím větší důraz na ochranu soukromí a osobních dat. Tlak na zvyšování této ochrany přichází nejen od uživatelů, ale i z nově přijímané legislativy jakou je v Evropské unii například GDPR (*General Data Protection Regulation*) a eIDAS (*Electronic Identification and Trust Services*), čehož si všímá práce [21]. Autentizační systém PEAS využívající knihovnu RKVAC je řešením, jak provádět spolehlivou autentizaci a zároveň zajistit soulad s požadavky na ochranou zmiňovaných dat. Byl z části představen v práci [22]. Systém PEAS se postupně vyvinul ze starších systémů, viz literatura [23], [24] a [25].

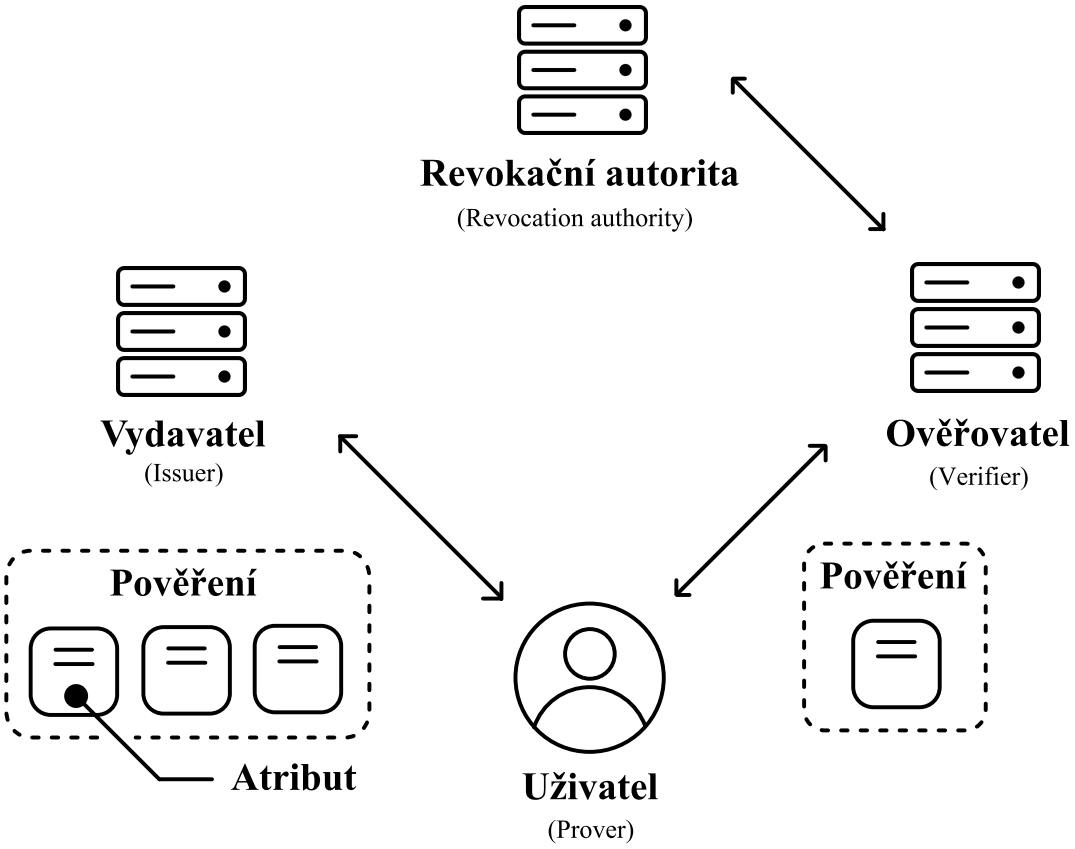
Oproti standardním autentizačním systémům přidává další kryptografické technologie, které kromě integrity, utajení, autentičnosti a nepopiratelnosti zajišťují navíc i **anonymitu**, **nespojitelnost** a **nesledovatelnost**.

Pro dosažení těchto požadovaných vlastností používá anonymní atributová pověření, jedná se tedy o atributový autentizační systém. Autentizace na základě těchto atributů dokáže skrýt identitu uživatele před ověřovatelem a zajistit nespojitost transakcí. To znamená, že uživatel odhalí pouze potřebné atributy, kterými mohou být například pracovní pozice, nebo pohlaví, ale zbytek jeho identity zůstane skryt. Nespojitost transakcí poté zajišťuje, že i opakované požadavky na autentizaci od stejného uživatele jsou pokaždé z pohledu ověřovatele rozdílné.

Tyto atributy jsou společně uzavřeny do skupiny, takzvaného pověření, které vydává uživatelům vydavatel a ručí za ně svým digitálním podpisem. V systému jsou definovány celkem čtyři entity.

- **Vydavatel** (*Issuer*) – zajišťuje vydávání digitálních pověření uživatelům.
- **Uživatel** (*User*) – je držitelem tohoto pověření a obsažených atributů, se kterými se anonymně prokazuje ověřovateli.
- **Ověřovatel** (*Verifier*) – ověřuje důkaz o držení atributů, které mu předkládá uživatel.
- **Revokační autorita** (*Revocation authority*) – revokuje uživatele ze systému.

Podrobněji se atributové autentizaci a kryptografickým metodám v technologii RKVAC věnují práce [23] a [22]. Následující schéma na obrázku 4.1 zobrazuje obecnou strukturu celého autentizačního systému a vzájemné komunikace jednotlivých entit.



Obr. 4.1: Obecné schéma autentizačního systému.

4.2 Požadavky na webové aplikace

V rámci praktické části této práce je zapotřebí navrhnout a implementovat celkem čtyři webové aplikace, jednu pro každou ze stran autentizačního systému. Obecně je požadováno zpřístupnit všechny funkce, které nabízí současná implementace autentizačního systému, uživatelům pomocí GUI založeného na webových technologiích.

Vydavatel (*Issuer*) – musí mít možnost přijímat požadavky na registraci od uživatelů, dále by měl spravovat databázi uživatelů a nařizovat jejich revokace.

Uživatel (*User*) – potřebuje rozhraní pro vytoření zmiňovaného požadavku na vydavatele a následně musí být schopen se s přidělenými atributy ověřit u ověřovatele. Výsledek této verifikace musí být graficky zobrazen uživateli.

Ověřovatel (*Verifier*) – jeho grafického rozhraní musí dovolovat vybírat, jaké atributy bude od uživatelů vyžadovat k předložení. Následně musí toto GUI zobrazovat i informace obsažené v přístupovém logu.

Revokační autorita (*Revocation authority*) – zde je jediným požadavkem informovat uživatele o událostech obsažených v logu. V něm jsou obsaženy výzvy na revokace jednotlivých uživatelů.

5 Soudobá pravidla pro tvorbu GUI

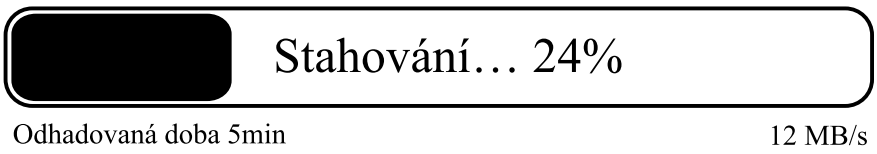
Uživatelské rozhraní neboli UI, je kritickou součástí každého softwarového produktu. Když je navrženo a implementováno korektně, uživatel o něm nepřemýšlí a soustředí svoji pozornost na produkt samotný. Když je však vytvořeno špatně, může tvořit překážku v komunikaci mezi uživatelem a produktem [26].

Pro dosažení kýženého výsledku při tvorbě rozhraní, většina designérů následuje několik principů. Následující pravidla jsou velmi zobecněné zásady, tak jak je formuloval Jakob Nielsen, Ph.D ve své práci [27]. Po jejich implementaci by měla vzniknout uživatelsky přívětivá aplikace. Většina z nich jde použít na všechny druhy interaktivních systémů od stolních počítačů po chytré hodinky.

Viditelnost systémového statusu

Grafické rozhraní by vždy mělo informovat uživatele, co se děje v systému běžícím pod ním a to v přiměřeně dlouhém čase. Když uživatel zná, v jakém stavu se systém momentálně nachází, je běžně tolerantnější něž uživatel bez této vědomosti. Následně se učí z následků svého konání v systému a odvíjí od něj své další chování.

Příkladem tak může být stahování objemného souboru ze vzdáleného serveru na klientův stroj, viz obrázek 5.1. Pokud systém nebude ukazovat indikátory, jakými jsou typicky procenta z již stažených dat, rychlost stahování a odhadovaný čas do úplného stažení, uživatel s vysokou pravděpodobností nebude čekat na dokončení tohoto procesu. Pokud je ovšem seznámen s těmito informacemi, může při své další interakci se systémem počítat s časovou náročností jednotlivých úkonů.



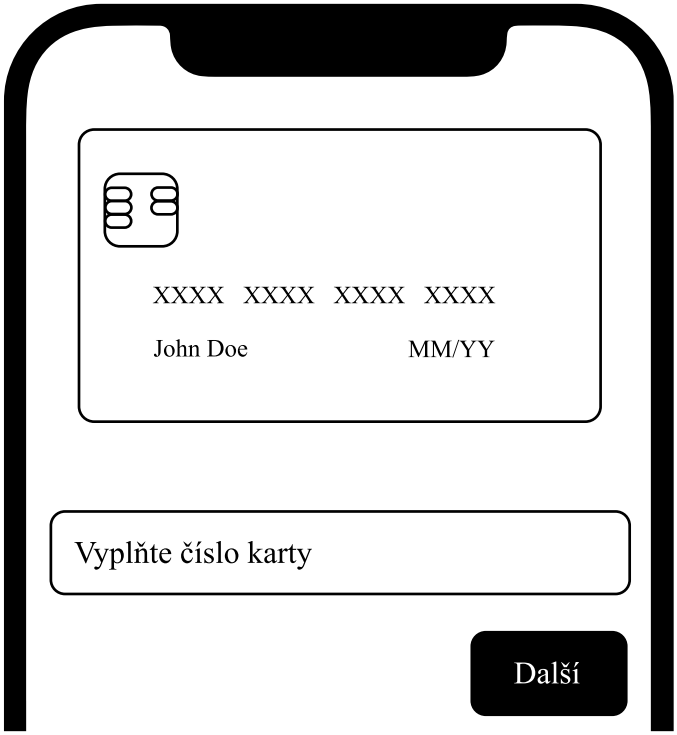
Obr. 5.1: Zprostředkování informací o stavu systému.

Přirovnání virtuálních konceptů k reálným předmětům

Je důležité, aby již design intuitivně naznačoval uživateli svoji funkci. Měl by imitovat koncepty, se kterými je uživatel seznámen z reálného světa a přirozeně jej provádět funkcemi systému. Jednou z nejčastějších designových metafor, kterou pozná každý uživatel počítače je koš. Všechny smazané soubory a dokumenty putují právě do něj, stejně jako již nepotřebné věci v reálném světě.

Takové metafory vytvářejí silné spojení mezi designovým vyjádřením a funkcí. Dovolují uživateli rychle využít znalosti z reálného světa a uplatnit je v tom virtuálním.

Dalším takovým příkladem může být výzva pro zadání čísla kreditní karty, jaká je zobrazena na obrázku 5.2. Pokud je v grafickém rozhraní znázorněna podoba reálné karty a zvýrazněna požadovaná čísla, je okamžitě uživateli jasné co je po něm požadováno.



Obr. 5.2: Spojení požadavku s reálným předmětem.

Uživatelská kontrola a svoboda

Uživatelé jsou lidé a dělají chyby. Je nutné nabídnout jim možnost vrátit akci, kterou buď vyvolali omylem anebo očekávali jinou reakci systému. Když je pro uživatele jednoduché anulovat svoji akci, vrátit se z procesu, umocňuje to v něm pocit jistoty a kontroly. Když se cítí, že má vše pod kontrolou, vybízí jej to k rychlejšímu a zábavnějšímu prozkoumávání ostatních funkcionalit systému. Nemá obavy ze své další akce.

Nejčastěji se lze setkat s tímto konceptem v textových editorech, kdy pomocí akce „Vrátit zpět“, nebo velmi známé klávesové zkratky `ctrl + Z` lze zrušit poslední změny v dokumentu.

Konzistence a standartizace

Uživatel by neměl přemýšlet nad významem jednotlivých výrazů a slov. Designéři by měli při tvorbě UI postupovat dle všeobecně známých standardů. Jakobovo[1] pravidlo říká, že lidé stráví většinu času v aplikaci někoho jiného. Při vývoji by měla být tato skutečnost respektována a designéři by se neměli snažit vymýšlet vlastní nové koncepty tam, kde jsou již zaběhnuta jiná pravidla.

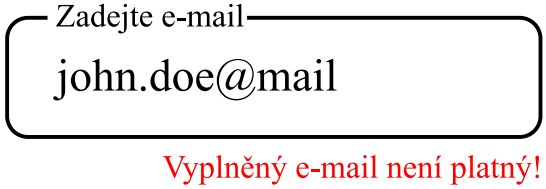
Takovým příkladem může být „košík“ v *e-commerce* prostředí. Uživatelé jsou zvyknutí, že nákupní košík je metaforou pro virtuální místo, kde se shromažďují vybrané produkty před jejich zakoupením. Pokud nebude tato konvence dodržena, nemusí být zcela jasné jak se v aplikaci orientovat.

Předcházení chybám

Dobré chybové zprávy jsou důležité, ale ještě lepší je předcházet samotnému vzniku chyb. Možností je hlídat uživatelské vstupy a posloupnost akcí.

Pokud je potřeba vyplnit například věk, měl by systém hlídat, že na vstupu je pouze číselná hodnota a nic jiného. V opačném případě je právě na grafickém rozhraní, upozornit na možnou chybu uživatele a předcházet tak nedovolené operaci, viz obrázek 5.3.

Existují dva základní druhy chyb, překlepy a vědomé chyby. Překlepy se stávají nevědomky z nepozornosti uživatele. Ty druhé způsobuje neporozumění uživatele modelu dat, který je po něm vyžadován.



Obr. 5.3: Upozorněním uživatele na neplatný vstup lze předcházet chybám.

Vědění raději než vzpomínání

Je pravidlo, dle kterého by co nejvíce elementů a prvků mělo být viditelných a nebo snadno přístupných uživateli. Snižuje to nároky na jeho krátkodobou paměť a tím zvyšuje rychlost orientace v aplikaci.

[1] Autorem je Jakob Nielsen, Ph.D, který je i autorem základních pravidel tvorby UI.

Flexibilita a efektivita

Reprezentací této filozofie je například již výše zmiňovaná klávesová zkratka `ctrl + Z`. Pro nové uživatele je to zcela skrytá a nijak nevyrušující funkcionalita. Pro zkušené uživatele velké zrychlení a zefektivnění práce se systémem.

Zároveň je zde zachována jistá flexibilita. Různým uživatelům mohou vyhovovat různé styly práce a záleží na nich, jaký ze způsobů si vyberou.

Estetický a minimalistický design

Měla by vždy existovat rovnováha mezi množstvím elementů na stránce a jejich informační hodnotou. Pokud je to možné, měl by se designér snažit v co nejmenším množství prvků na stránce sdělit co největší množství informací.

K tomuto přístupu by měla být komplementem jednoduchá s stručná grafika. Neznamená to nutně povinnost používat pouze jednoduché tvary, ale především nevytvářet grafický smog.

Pomáhat uživatelům diagnostikovat a zotavit se z chyb

Chybové hlášky by měly být vyjádřeny srozumitelným jazykem a podány tak, aby jim běžný uživatel mohl porozumět. V ideálním případě přesně popsat jaký problém se vyskytl a navrhnout řešení. Tato hlášení by měla být doprovázena i grafikou, aby byla pro uživatele snadno rozpoznatelná.

Použití chybových kódů je možné, ale neměly by být uživateli viditelné, nebo umístěny na málo prominentním místě.

Nápověda a dokumentace

V nejlepším případě by neměla být dokumentace při používání produktu potřeba, je však dobré mít oporu pro okamžik, kdy uživatel nedokáže sám dokončit požadovanou úlohu. Dokumentace by měla pomoci uživateli porozumět, jak tento úkol dokončit a popsat přesné kroky požadované k dosažení daného cíle.

Více o těchto pravidlech lze najít v článku [28], nebo v práci [29], která je porovnává s dalšímy pravidly od jiných autorů.

6 Návrh a implementace webových aplikací

Kapitola prezentuje výsledky a poznatky praktické části bakalářské práce. Ty jsou tvořeny především implementovanými webovými aplikacemi ve zvoleném frameworku Vue.js a doplňujícími webovými servery, které zajišťují propojení těchto aplikací s backendem. Uvedené aplikace i servery jsou součástí elektronické přílohy této práce.

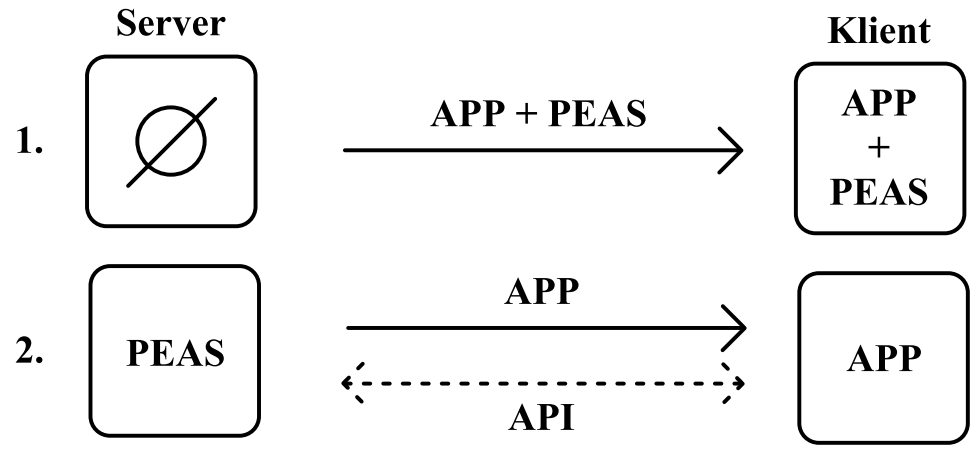
V první části 6.1 je pojednáváno o dvou návrzích architektury systému a o tom jaká a proč byla nakonec zvolena. Navazující téma 6.2 poté přibližuje podrobněji jednotlivé komponenty této architektury. Poslední významnou částí bylo navržení loga a grafiky pro celý uvažovaný systém. Viz 6.3.

6.1 Architektura systému

V této podkapitole je vysvětleno logické rozložení hlavních komponent systému. Jsou zde srovnávány dvě možné varianty provedení, ze kterých bylo vybíráno. První z nich cílí na vytvoření komplexního programu, který by se skládal z webové aplikace a programu PEAS, kde by následně tento celek byl distribuován klientům v podobě webu.

Druhá varianta přichází s tradičnějším řešením. Nechává výpočetní část, tedy program PEAS na straně serveru a distribuuje klientům pouze frontend, který následně komunikuje se serverem pomocí API.

V závěru každé z podkapitol jsou zhodnoceny klady a zápory těchto variant. Následující schéma 6.1 demonstruje ve velmi zobecněném pojetí základní rozdíly mezi těmito přístupy.



Obr. 6.1: Zobecněné znázornění rozdílů mezi první a druhou variantou.

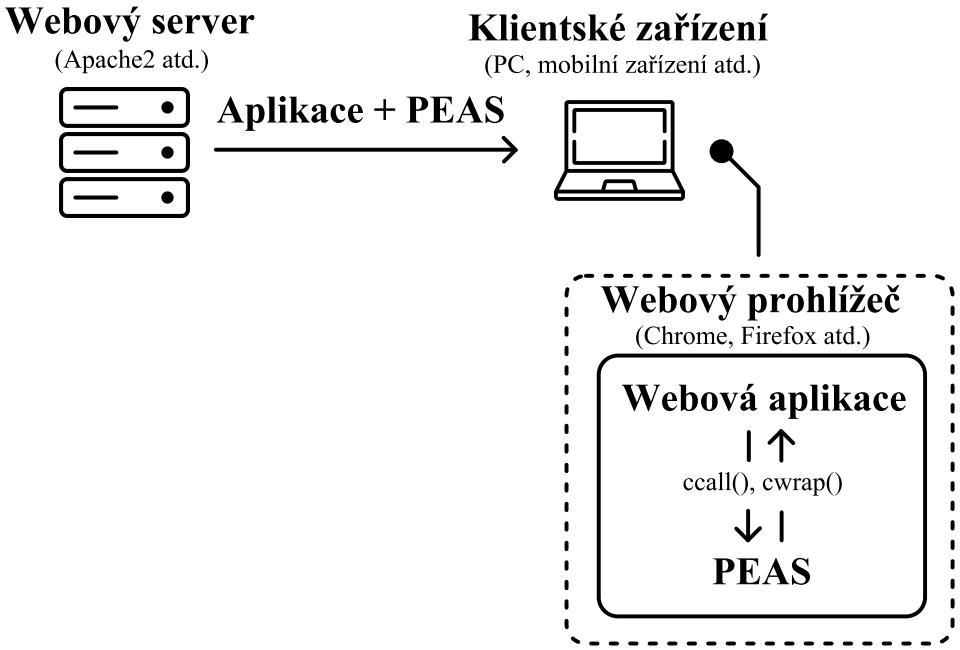
6.1.1 Varianta první – WebAssembly

Teoretická polovina práce je z části zaměřena právě na analýzu proveditelnosti této varianty řešení, tedy vložit celý program PEAS včetně kryptografické knihovny RKVAC do webové aplikace. K tomu měla pomoci zmiňovaná technologie *WebAssembly*, jejíž podstatou bylo přemostit komunikaci mezi webovou aplikací používající JavaScript a programem PEAS používající jazyk C.

Webová aplikace

V této variantě by výsledným produktem byla webová aplikace, kterou by bylo možné umístit na statický webový server, jakým je například *apache2*. Ten by byl zodpovědný pouze za distribuci těchto aplikací do prohlížečů klientů.

Jelikož by stažená webová aplikace již sama o sobě obsahovala veškerou logiku, reprezentovanou programem PEAS, nemusela by se spoléhat na žádný *server-side* backend. Veškeré kryptografické výpočty by byly prováděny přímo na klientském zařízení a pouze výsledek by byl komunikován dalším stranám systému, jakou je například ověřovatel (*Verifier*). Výměna informací v rámci aplikace by probíhala pomocí funkcí `ccall` a `cwrap`, které nabízí technologie *WebAssembly*. Viz 3.2.1. Ty dokáží zavolat funkce obsažené v programu PEAS a začlenit ho tak přímo do webové aplikace. Na obrázku 6.2 je schématicky naznačena obecná struktura distribuce jedné webové aplikace v této variantě.



Obr. 6.2: Schématické znázornění varianty s WebAssembly.

Shrnutí

Výhodou této varianty jsou malé nároky na infrastrukturu. Jelikož uživatelský frontend komunikuje přímo s programem PEAS, který je jeho nedílnou součástí a nevyžaduje přenášení uživatelských dat po síti k serveru a zpět. Tím pádem je i o něco méně závislý na připojení k síti.

Nevýhody jsou však značné. První a nejdůležitější je bezpečnost, jelikož je celý program a jeho kryptografické jádro, i když pouze v binární podobě umístěno na klientském zařízení, kde je potenciální hrozbou přístup útočníků k tomuto kódu. Dále jsou zde kladeny zvýšené nároky na klientská zařízení, která zároveň provádí i výpočty. Každá aktualizace programu PEAS by v této variantě musela také nutně znamenat i aktualizaci celé webové aplikace, právě kvůli těsnému provázání těchto dvou komponent. To je další nevýhoda, která by mohla zpomalovat proces vývoje.

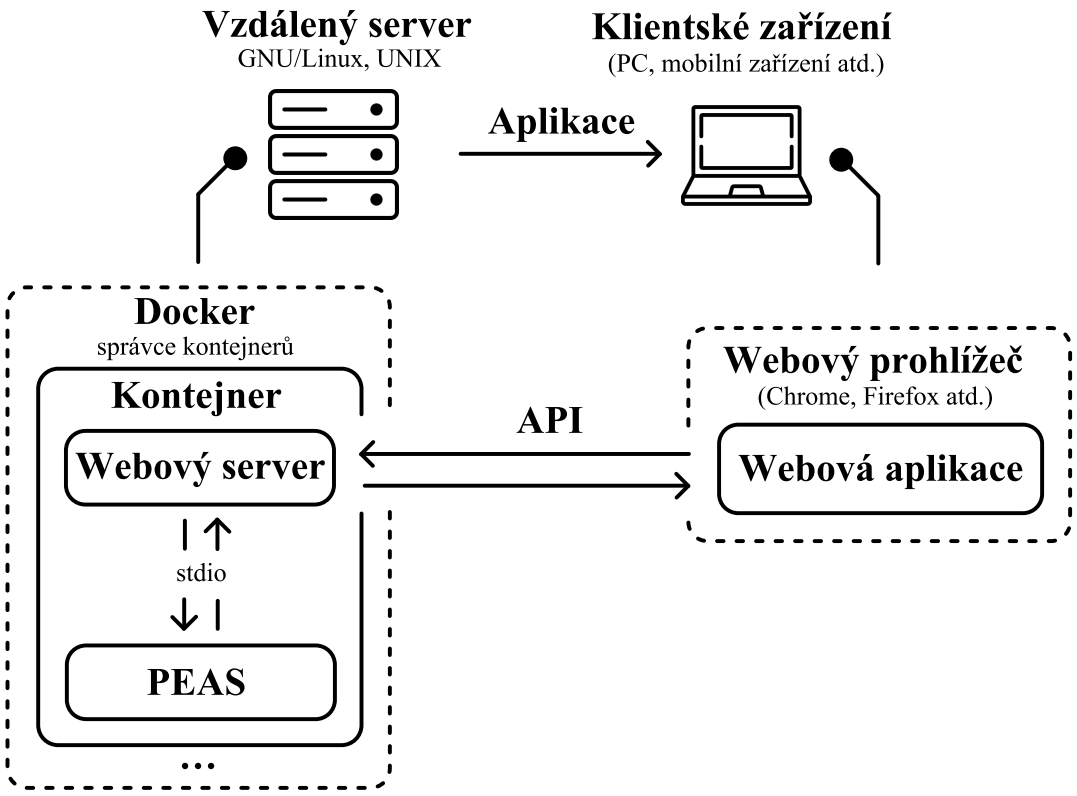
Komplikovaná je i samotná realizace tohoto způsobu provedení. Jak již blíže představil oddíl 3.2.1 pro začlenění programu psaném v C do webové aplikace je potřeba jeho kompilace speciálním kompilátorem a vytvoření modulu. Tento proces není pro jednoduchý program nikterak komplikovaný, ale se současnou komplexitou programu PEAS a knihovny RKVAC, kdy tyto celky mají velký počet dalších závislostí na nestandartních knihovnách, bylo téměř nemožné v krátkém čase vytvořit fungující prototyp. Pro převažující zápory v této metodě od ní bylo nakonec upuštěno a výsledné řešení používá architekturu blíže popsanou v následující podkapitole.

6.1.2 Varianta druhá – REST API

Druhou nabízející se variantou řešení je použít pro přemostění komunikace REST API. Tato varianta je v mnoha ohledech diametrálně odlišná od varianty prvé, ale používá strukturu, která je více blízká tradičnímu pojetí webových aplikací. Schématicky tuto variantu znázorňuje obrázek 6.3.

Webová aplikace

V první řadě v tomto provedení již není součástí aplikace program PEAS, který je přesunut na stranu serveru. Tím došlo k jasné separaci frontedu a backendu. Klient tedy po zadání adresy webové aplikace do prohlížeče obdrží pouze ji. Ta následně slouží jako komunikační nástroj, překládající požadavky uživatele zadané skrz grafické rozhraní na API volání. Ty již ale musí obsloužit server, který disponuje instancí programu PEAS. Podrobnosti k implementaci a použitým technologiím obsahuje sekce 6.2.1.



Obr. 6.3: Schématické znázornění varianty s REST API.

Webový server

Hlavní překážkou v provedení této varianty řešení je neexistující podpora programu PEAS pro komunikaci pomocí takového API. Součástí práce bylo tedy i vyvinutí vlastního webového serveru pro každou z aplikací, který slouží jako pomyslný prostředník mezi touto aplikací a programem PEAS. Jeho hlavním úkolem v komunikaci, je překládání API dotazů na příkazy předávané programu PEAS pomocí standardního vstupu `stdin` a opětovné odesílání výstupů přijatých na rozhraní `stdout` zpět pomocí API odpovědí aplikacím. Více v oddílu 6.2.2.

Kontejnery Docker

Pro usnadnění jak správy, tak nasazení celého systému na vzdálený server jsou využity kontejnery Docker, které obsahují instalaci programu PEAS a připravený webový server s odpovídající aplikací. Pro každou stranu systému náleží jeden kontejner. Každý z kontejnerů mapuje jeden z portů hostujícího systému na vlastní port 80 pro zpřístupnění zmiňovanému webovému serveru v lokální síti hostujícího zařízení. Ostatní komunikace mezi kontejnery a tedy i entitami systému je obsluhována sítí Docker.

Shrnutí

Ve výsledku tedy tato varianta připomíná svou stavbou typickou webovou stránku. Zároveň odstraňuje i velkou část závad, kterými byla zatížená předchozí navrhovaná architektura popsaná v sekci 6.1.1. Jak již bylo zmíněno, dochází zde ke striktnímu rozdělení backend a frontend komponentů. To umožňuje tyto komponenty relativně nezávisle na sobě upravovat. Celá výpočetní a kryptografická část je umístěna v kontrolovaném prostředí na vzdáleném serveru. Klient disponuje pouze webovou aplikací. Tento model je jak bezpečnější, tak klade menší nároky na hardware klienta.

Za nevýhodu by se v této variantě dala označit velká závislost na připojení k serveru. Pokud dojde k jeho přerušení, uživatel je zanechán pouze s frontendem, který nedisponuje téměř žádnou offline funkcionalitou.

6.2 Komponenty

Tato podkapitola se podrobněji věnuje komponentům systému a jejich implementaci po technické stránce. Vysvětluje detaily zvažované při jejich vývoji. Dělí se na dvě podsekce. Jelikož výstupem práce jsou **webové aplikace** a jejich podpůrné **webové servery**, byl každé z částí věnován samostatný oddíl. Třetím často zmiňovaným komponentem je **program PEAS**, pro který je nadstavba v podobě serverů a aplikací určena. Ten je ale vyvíjen odděleně a není předmětem této práce.

6.2.1 Webové aplikace

Webové aplikace jsou stěžejním komponentem a jádrem celé práce. Celkem byly vytvořeny čtyři webové aplikace, jedna pro každou entitu figurující v autentizačním systému PEAS. Viz kapitola 4.

Následující text dělí jejich vývoj do tří tématických celků. Prvním je vzhled. Ten je zde velice důležitý, jelikož webová aplikace představuje frontend, který má za úkol nejen poskytnout uživateli grafické rozhraní, ale i reprezentovat celý systém stojící za ním. Další sekce se věnuje těm nejdůležitějším způsobům jakými aplikace interaguje s uživatelem. Nakonec jsou zde zmíněny technické a implementační detaily, kterými autor přispěl nad rámec frameworku.

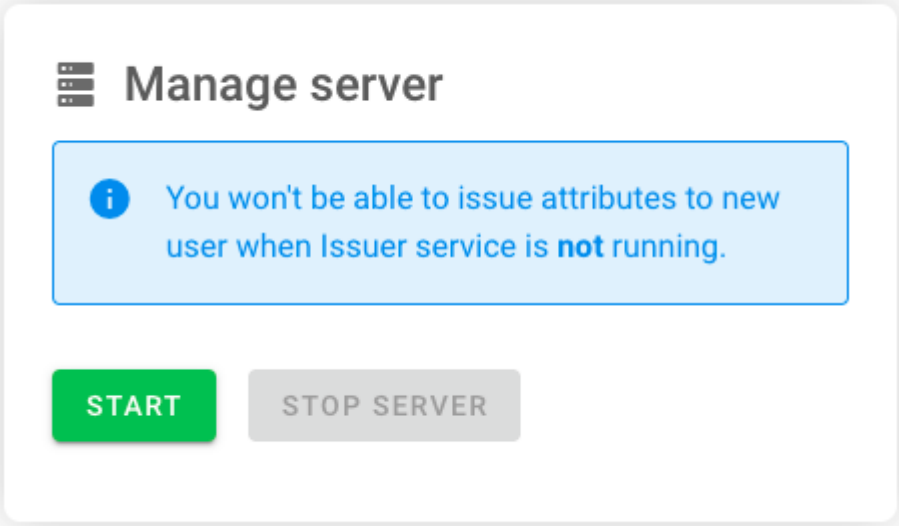
Vzhled

Základními parametry, které rozhodovaly při návrhu grafického uživatelského rozhraní byly **jednoduchost**, **intuitivnost** a **robustnost**. Jelikož se jedná v první řadě o rozhraní, které má umožňovat řízení systému, je potřeba jednoduše zpřístupnit všechny funkce uživateli tak, aby si je intuitivně dokázal osvojit. Robustní v

tomto kontextu znamená, že aplikace by se měla chovat dle představ uživatele v jakémkoliv stavu i při nestandartních situacích a zacházení.

Všechny prvky, které lze najít v aplikacích, včetně loga, následují standard *Material Design*, který je osvědčenou volbou pro tvorbu moderního GUI [31]. Tento směr byl poprvé představen společností Google na konferenci I/O v roce 2014 a od té doby byl masivně adoptován. Naplňuje tak pravidlo konzistence a standardizace popsané v kapitole 5.

Nejvýraznějším prvkem tohoto designu je motiv karet, který se odráží i v aplikacích PEAS. Jednotlivé aplikace by měly působit jako soubory kartiček rozprostřené před uživatele. Každá z nich má své vlastní poslání v rámci aplikace. Jedna tedy například agreguje ovládací prvky serveru, jiná nabízí nápovědu apod. Praktická implementace karty je vidět na následujícím obrázku. 6.4



Obr. 6.4: Základním stavebním kamenem je motiv karty.

Barvy použité v aplikaci následují standardy definované v grafickém manuálu, který byl zpracovaný jako součást tvorby loga a je dostupný v příloze této práce. Primární barvou je zelená, která je tématicky spojena s logem a názvem PEAS, který v překladu znamená hrášek. Všechny barvy využívají své živější odstíny, čímž následují moderní směr, kterým se ubírají i ostatní uživatelská rozhraní.

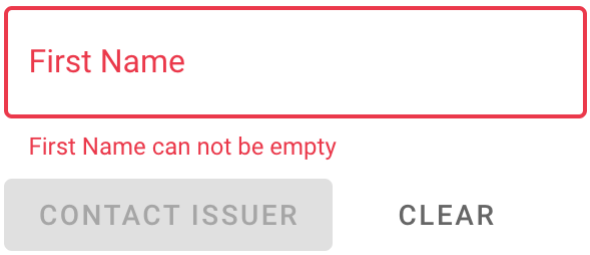
Statické obrázky v aplikacích byly tvořeny jako součást jejich vývoje. Animované ilustrace byly naopak převzaty a upraveny pod volnou licencí CC-BY z komunitního portálu LottieFiles[1].

[1]Více viz https://lottiefiles.com/page/license

Interakce s uživatelem

Navržené webové aplikace se snaží řídit a kontrolovat interakce uživatele tak, aby mohly nabídnout pokud možno tu nejlepší UX. Implementují pokročilé metody kontroly uživatelských vstupů, informují jej srozumitelně o příznivých i nepříznivých událostech v systému, a to vše za použití grafických elementů.

Kontrola vstupů, které zadává uživatel je, jak popisuje kapitola 5, jedním z kritických míst. Proto již webová aplikace dohlíží na správný tvar vkládaných dat ještě předtím, než jsou poslány ke zpracování na server. K tomu využívá rozšíření dostupné v balíčku `@vee-validate`. Pomocí jeho funkcí je ošetřeno každé textové pole. Znaky zadávané uživatelem jsou tedy kontrolovány předem definovanými pravidly a v případě nesplnění jejich nároků je uživatel konfrontován s upozorněním, které lze vidět na obrázku 6.5.



Obr. 6.5: Kontrola vstupu od uživatele.

Všechny aplikace jsou vybaveny jednotným systémem upozornění. Toho využívá především API služba, která v případě neúspěšné operace pomocí této notifikace upozorní uživatele. Více v následujícím oddíle. Druhá velmi důležitá služba využívající tuto notifikaci k upozornění uživatele, je služba hlídající spojení mezi serverem a webovou aplikací. V případě přerušení spojení je aplikace uvedena do módu offline a uživatel notifikován. Tato situace je demonstrována následujícím obrázkem 6.6.

Obr. 6.6: Jednotná notifikace pro upozornění uživatele.

Technické provedení

Pro samotnou realizaci webových aplikací byl použit framework Vue.js, jenž byl vybrán v porovnání s dalšími vývojovými prostředími v teoretické části této práce, viz kapitola 2. Pomocí něho byla vytvořena kostra projektu každé z aplikací a přidána další rozšíření pomocí Vue CLI, které jsou uvedené v následující tabulce.

Tab. 6.1: Přehled hlavních přidaných závislostí do projektů webových aplikací.

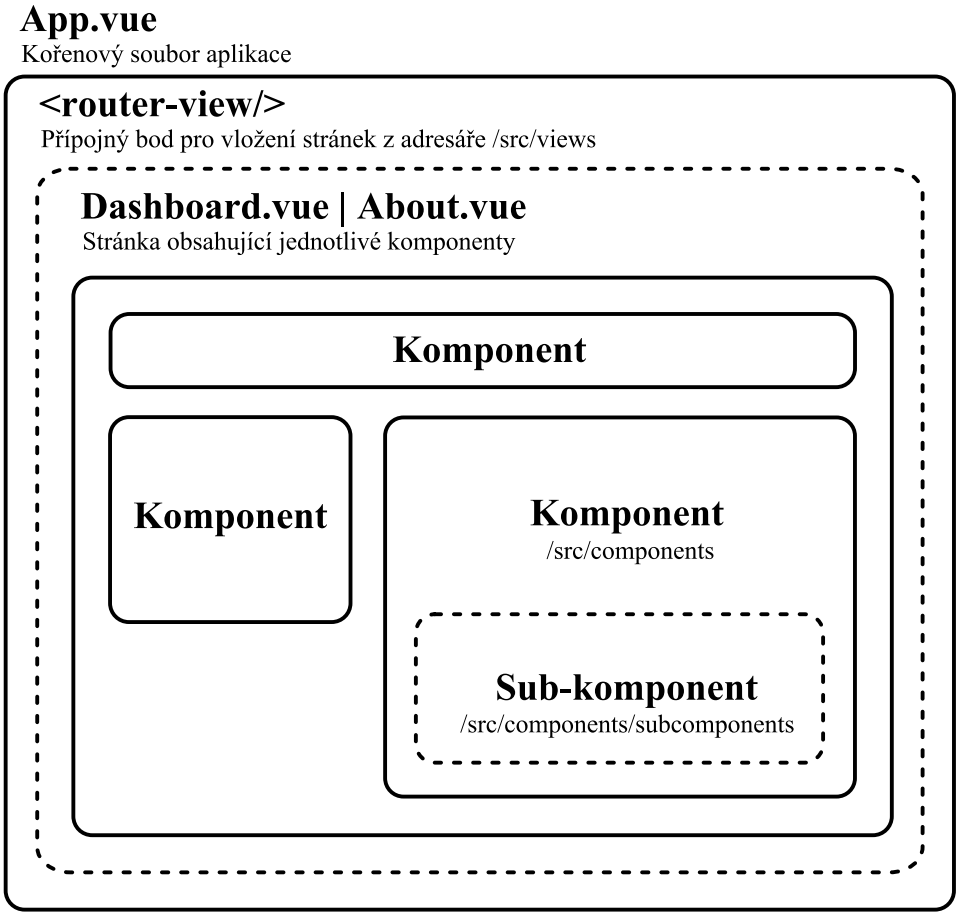
Název	Balíček	Popis
PWA	@vue/cli-plugin-pwa	Podpora funkcí Progressive Web Application.
Router	@vue/cli-plugin-router	Přidání možnosti routování mezi stránkami.
Vuex	@vue/cli-plugin-vuex	Nástroj pro správu globálních proměnných.
Vuetify	vue-cli-plugin-vuetify	Knihovna UI elementů pro Vue.js.
VeeValidate	@vee-validate	Validační nástroj pro formuláře.
LWV	lottie-web-vue	Komponent pro přehrávání animací.

Nejvíce využívaným rozšířením z výše uváděných je `vue-cli-plugin-vutify`. Jedná se o knihovnu UI komponentů vytvořenou právě pro framework Vue. Obsahuje kompletní sadu HTML komponent od jednotlivých tlačítek po pokročilé formuláře. Využívá vlastního globálně definovaného CSS. Pro webové aplikace PEAS, byly však tyto výchozí hodnoty upraveny a rozšířeny o vlastní globální styly. Do projektu je tato knihovna automaticky integrována pomocí Vue CLI. Následně lze začít využívat katalog předpřipravených komponent.

Jak již bylo více přiblíženo v teoretické kapitole 2, webové aplikace tvořené pomocí frameworku Vue.js jsou sestaveny ze znovu použitelných komponent. To přidává na modularitě a rychlosti vývoje, přičemž zároveň minimalizuje duplicity v kódu. Následující obrázek 6.7 se snaží schématicky zobrazit základní logickou strukturu webových aplikací.

Za stejným účelem, tedy zamezením zbytečné duplicity kódu byla vyvinuta i služba PEAS API Service, nacházející se v adresáři `/scr/API` každé z aplikací. Jejím úkolem je nabízet všem komponentům aplikací jednoduché rozhraní pro obsluhu API volání. Lze ji jednoduše importovat v místě potřeby z výše uvedeného umístění. Posléze je možné využívat její službu pomocí jednoduchého rozhraní. To je tvořeno následující funkcí `apiservice(<endpoint>, <input>)`.

Samotná služba se poté stará o kontaktování požadovaného koncového bodu API, specifikovaného pomocí `endpoint` URL. Parametr `input` může nabývat libovolného datového typu, nejčastěji však typu objekt. V případě, že neexistují žádná data k přenosu na server v rámci požadavku, předává se do parametru hodnota `null`.



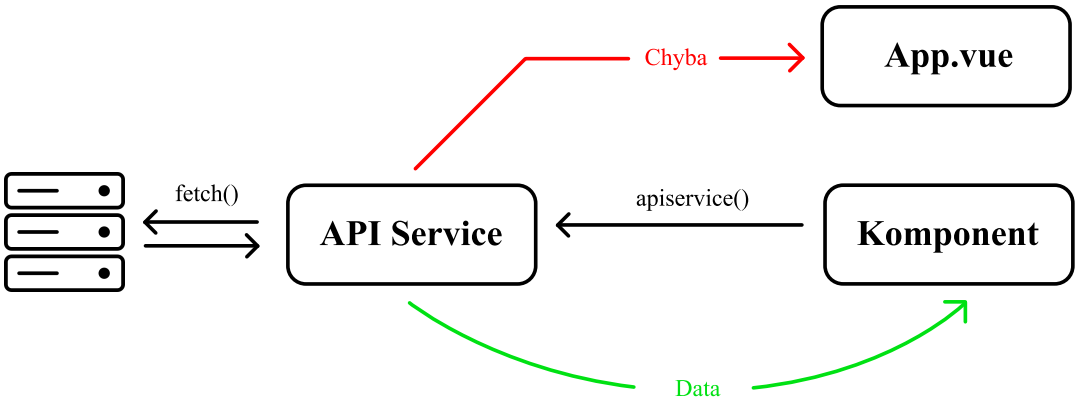
Obr. 6.7: Hierarchie provázání zdrojových souborů v projektu.

Před samotným voláním je definována hlavička a typ přenosu. Případná data jsou převedena do formátu JSON.

Další neméně užitečnou vlastností této služby je, že se sama v případě chyby postará o její zachycení a zpracování. Jak v případě, kdy se nepodaří vůbec navázat spojení se serverem, tak v případě že server vrátí chybový kód. Dokáže tuto skutečnost zaznamenat a vyvolat chybový event. Viz obrázek 6.8. Ten je propagovaný včetně přiložené chybové zprávy až do souboru App.vue, odkud je notifikací 6.6 informován uživatel. Není potřeba tedy při využívání této služby myslet na zachycování chybových stavů a přidávání dalšího kódu jakým je například *try-catch* blok.

6.2.2 Webové servery

Zastávají roli prostředníka v komunikaci mezi webovou aplikací a programem PEAS, nad kterým tvoří jakousi nadstavbu. Připojují se na jeho `stdio` rozhraní a předávají informace mezi ním a API *endpointy*.



Obr. 6.8: Schéma komunikace za pomocí API Service.

Technické provedení

Všechny servery byly pro jednoduchost vytvořeny ve stejném jazyce jako aplikace, tedy JavaScriptu. Jako jejich základ byl použit Express server framework, který je volně dostupný jako *npm* balíček. Základním posláním serverů je samozřejmě poskytovat klientům webové aplikace. Ty ale vyžadují zmiňované API rozhraní, které muselo být vytvořeno a přidáno do těchto serverů. Více v následujícím oddílu síťové komunikace.

Stěžejním úkolem při implementaci webových serverů bylo zajištění efektivní a spolehlivé komunikace s programem PEAS pomocí `stdio`. Program PEAS je designovaný pouze jako konzolová aplikace a jeho jediným rozhraním, které bylo možné využít pro vkládání vstupů od uživatele, bylo tedy CLI.

Runtime environment Node.js ve kterém je spuštěný každý server nabízí dvě funkce pro ovládání externích programů a jejich `stdio` rozhraní.

- **exec()** – využívá datový buffer.
- **spawn()** – využívá datový stream.

Jsou součástí modulu `child_process`. Z důvodu *single-threaded* povahy Node.js je zapotřebí nejprve vytvořit *child process* a ten až následně spouští externí program. Největším rozdílem těchto dvou funkcí je formát, ve kterém vracejí data. Funkce `exec()` spustí příkaz, který jí byl předán jako parametr, počká na jeho ukončení a poté vrátí data. Naopak `spawn()` začíná vracet data okamžitě, tak jak jsou generována programem v podobě streamu. V implementaci byly uplatněny obě metody.

Funkce `exec()` – v serverech je tato funkce využívána na spouštění příkazů u kterých je předpokládán pouze relativně malý výstup a lze čekat nějakou dobu na jeho vrácení. Použití *bufferu* totiž značně omezuje přijatelnou velikost výstupních dat. Po naplnění kapacity *bufferu* dochází k jeho přetečení. Typicky ji tedy využívá strana uživatele (*User*), jelikož nespouští žádný dlouhodobě vykonávaný proces, ale

pouze krátké příkazy, u kterých je rozsach vracených dat malý.

Výpis 6.1 demonstruje jak server využívá funkce `exec()` ke komunikaci s programem PEAS pomocí `stdio`. Funkci jsou předány dva parametry. Prvním je *string* obsahující požadovaný příkaz k provedení. Druhým je *callback* funkce, která přejímá jako své parametry výstupy z prováděného příkazu. Proměnná `error` je naplněna v případě, že dojde k chybě ještě před spuštěním samotného příkazu. Dále `stdout` a `stderr` obsahují data generované spuštěným příkazem ze stejnojmenných rozhraní, tedy standardního výstupu a standardního erroru.

Následně posledním úkolem *callback* funkce je zjistit jaký *buffer* byl naplněn a provést s ním definované operace, nejčastěji odeslat vrácená data pomocí API zpět do aplikace. Tím je celý proces komunikace s programem PEAS dokončen.

```
1   /* Import funkce 'exec' z child process. */
2   const { exec } = require("child_process");
3   /* Vykonání příkazu. */
4   exec('./peas_ue␣-i', (error, stdout, stderr) => {
5       /* Výpis chyby při spouštění programu. */
6       if (error) {
7           console.log(`error: ${error.message}`);
8       }
9       /* Výpis chyby programu. */
10      if (stderr) {
11          console.log(`stderr: ${stderr}`);
12      }
13      /* Výpis dat ze stdout. */
14      console.log(`stdout: ${stdout}`);
15  });
```

Výpis 6.1: Příklad syntaxe funkce exec().

Funkce `spawn()` – je ideálním nástrojem pro správu programů PEAS, které jsou typu server, tedy po spuštění nevrátí data a neskončí, ale začnou naslouchat dotazům na síti. Proto v tomto případě nelze použít předcházející funkci využívající *buffer*, ale je nutné využití této alternativy, která průběžně vrací data pomocí *streamu*.

Syntaxe je v tomto případě komplexnější i z důvodu zapojení standartního vstupu `stdin`, který v předchozí funkci nebyl zapotřebí. Nejkomplexnější implementace této funkce je použita ve webovém serveru strany vydavatele (*Issuer*), jejíž část je uvedena ve výpisu 6.2.

Ve zvolené implementaci jsou funkci `spawn()` opět předávány dva parametry, kdy prvním je *string* s příkazem který má být spuštěn. Druhým je seznam, který obsahuje případné další přepínače náležící ke spouštěnému příkazu. Jako třetí parametr

lze také přidat nastavení pro jednotlivé části rozhraní `stdio`. V něm lze definovat, v jakém módu budou tato rozhraní *child procesu* spojena s rozhraními hlavního procesu. Implementace použita v této práci využívá výchozího režimu `pipe`, tím pádem není zapotřebí dalšího nastavení.

Instance celé funkce je uložena do proměnné `peas` se kterou lze dále pracovat. Jelikož spouštěný program `peas_ie` je typu server, nelze na výstupní data čekat do jeho ukončení, které může nastat až po neomezeně dlouhém časovém úseku. Proto za pomoci zmiňované instance lze přistupovat ke dvěma výstupním streamům `stdout` a `stderr`. Funkce `on()` hlídá výskyt událostí (*eventů*) v těchto streamech. Pokud jsou spuštěným programem PEAS vypisována data do jednoho ze streamů, je v odpovídajícím streamu vyvolán *event* `data`. Ten zachytí funkce `on()` a samotná data předává jako vstupní parametr *callback* funkci, která s nimi může dále provádět požadované operace. Viz výpis 6.2.

Pokud je funkce `on()` zavolána přímo na instanci, může zachycovat i události typu `error` a `close`. První je ekvivalentem proměnné `error` u funkce `exec()` a je vyvolán v případě že se nepodaří vykonat definovaný příkaz. Druhá událost nastává při ukončení příkazu a jako data vrací kód typu *integer* se kterým byl příkaz ukončen.

```
1   /* Vytvoření instance programu v proměnné 'peas'. */
2   const peas = spawn("./peas_ie", []);
3   /* Naslouchání událostem v datových streamech
4   ** a na samotné instanci pomocí funkce 'on'. */
5   peas.stdout.on("data", data => {
6       console.log(`stdout: ${data}`);
7   });
8   peas.stderr.on("data", data => {
9       console.log(`stderr: ${data}`);
10  });
11  peas.on('error', (error) => {
12      console.log(`error: ${error.message}`);
13  });
14  peas.on("close", code => {
15      console.log(`PEAS Issuer service
16      exited with code ${code}\n`);
17  });
```

Výpis 6.2: Příkld syntaxe funkce spawn().

Výše uvedené funkce tedy umožnily komunikaci směrem od programu PEAS. Dalším řešeným problémem bylo předávání vstupních dat do programu. V případech, kdy je program spouštěn a již známe potřebná vstupní data, jsou mu jednoduše

poskytnuta v rámci příkazu. Příkladem může být spuštění programu PEAS pro stranu vydavatele (*Verifier*), kdy je jako vstup očekáván přepínač specifikující čísla atributů, které ověřovatel požaduje k předložení. Při využití funkce `spawn()` lze tuto situaci zapsat následovně. Viz 6.3.

```
args = '-d␣1,2'
const peas = spawn("./peas_ve", [args]);
```

Výpis 6.3: Ukázka předání vstupních dat při spuštění programu.

Toto řešení ale není možné aplikovat v případě, že je nejprve potřeba spustit požadovaný program a až následně je zahájeno interaktivní dotazování na vstup. Tento scénář nastává při spuštění programu PEAS strany vydavatele (*Issuer*). Ten je typu server a naslouchá na síti příchozím požadavkům od klientské aplikace uživatele (*User*). Po zachycení požadavku je spuštěno konzolové dotazování na vstupní data, které je při použití pouze s CLI rozhraním programu ručně vyplněno uživatelem.

Hlavním cílem bylo tedy automatizovat vkládání požadovaného vstupu pro použití s GUI. Toho lze docílit opět pomocí rozhraní, které nabízí funkce `spawn()`. Po vytvoření instance této funkce v proměnné `peas`, lze přistoupit tentokrát ke vstupnímu streamu `stdin`. Data jsou poté vkládána do fronty na vstup programu pomocí funkce `wait()`, jako její její parametr. V následujícím výpisu 6.4 je úryvek kódu znázorňující implementaci této funkce ve webovém serveru. Do fronty na vstup programu jsou vkládána data obdržená v rámci API od webové aplikace. Jednotlivé údaje jsou proloženy symbolem `\n`, vkládajícím nový řádek a tím simulující úhoz *enetru*, jinak očekávaného od uživatele. Více o širším začlenění této funkce v následujícím oddílu.

```
peas.stdin.write(`${req.body.name}\n
                  ${req.body.id}\n
                  ${req.body.employer}\n
                  ${req.body.position}\n`);
```

Výpis 6.4: Ukázka předání vstupních dat při běhu programu.

Síťová komunikace

Při tvorbě webových aplikací pomocí varianty využívající REST API, bylo zapotřebí definovat další síťovou komunikaci, která doplňuje výměnu informací mezi programy PEAS (Viz obrázek 6.10 čárkovaně.) o komunikaci mezi aplikacemi a servery. Původní síťová komunikace mezi programy PEAS je zprostředkována pouze vnitřní sítí Docker. Aby bylo možné pro uživatele využít webové aplikace, je ale zapotřebí zpřístupnit webový server každého kontejneru na hostujícím zařízení. Všechny servery naslouchají na portu 80, který je ale dostupný pouze v rámci daného kontejneru.

Proto za pomocí *docker bridge* jsou ve výchozím nastavení tyto porty 80 mapovány na různé porty hostujícího zařízení, viz tabulka 6.2.2.

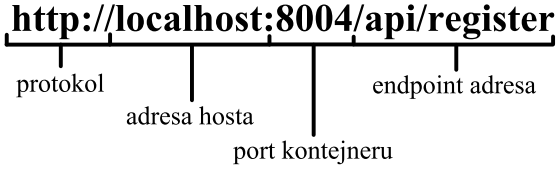
Tab. 6.2: Přehled mapování portů.

Kontejner	Mapování
Revokační Autorita	:80 → :8001
Uživatel	:80 → :8002
Ověřovatel	:80 → :8003
Vydavatel	:80 → :8004

API – je hlavním způsobem komunikace mezi webovými aplikacemi a servery. Aplikace využívají zmiňované PEAS API Service, která implementuje vestavěnou funkcionalitu JavaScriptu, tedy funkci `fetch()`. Ta přijímá dva parametry v podobě URL adresy koncového API, neboli *endpointu* a objekt *options* obsahující hlavičku požadavku a přenášená data. Tato struktura byla popsána v podkapitole 3.1.

Na straně serveru je proto potřeba poskytnou aplikacím tyto API endpointy. Tvoří je funkce `post()`, její první parametr definuje adresu na níž tento endpoint bude naslouchat a druhým parametrem je opět *callback* funkce. Ta přejímá parametry `req` a `res` obsahující příchozí a odchozí data. V těle této funkce jsou definovány požadované operace, které má server s příchozím požadavkem provést.

Následující obrázek 6.9 znázorňuje typickou URL *endpointu* používaného v implementaci. Jde o vlastní specifikaci, kdy veškeré adresy náležící API jsou označeny adresářem `/api/` pro jejich jednoznačnou identifikaci. Samotné adresy se dělí na několik částí, přičemž nejdůležitější je poslední oddíl *endpoint adresa*. Ten lze uvést při volání z aplikace i samostatně, pokud je tento *endpoint* umístěn na serveru ze kterého je hostována i webová aplikace. Zbylá část adresy je poté automaticky doplněna doménovým jménem aplikace. Při kontaktování jiného serveru, je ale zapotřebí tuto adresu uvést v celém formátu.



Obr. 6.9: Schéma adresy endpointu.

Odpověď na API dotazy je vždy strukturována dle následujícího klíče. Je odesílán celý objekt, který vždy obsahuje dvě proměnné. První je `code`, která nabývá v

případě úspěšně provedeného dotazu a operace hodnotu 200. Pokud dojde k chybě je vrácena hodnota 500. Druhá proměnná se nazývá `data` a obsahuje vždy vrácená data po operaci, nebo v případě nezdaru *string* s chybovou hláškou, která bude zobrazena uživateli. Poslední možností je, že bude mít hodnotu `null` pokud je operace úspěšná, ale neexistují data pro přenos.

```
res.json({code: 200, data: null});
```

Výpis 6.5: Jednotná struktura API odpovědí.

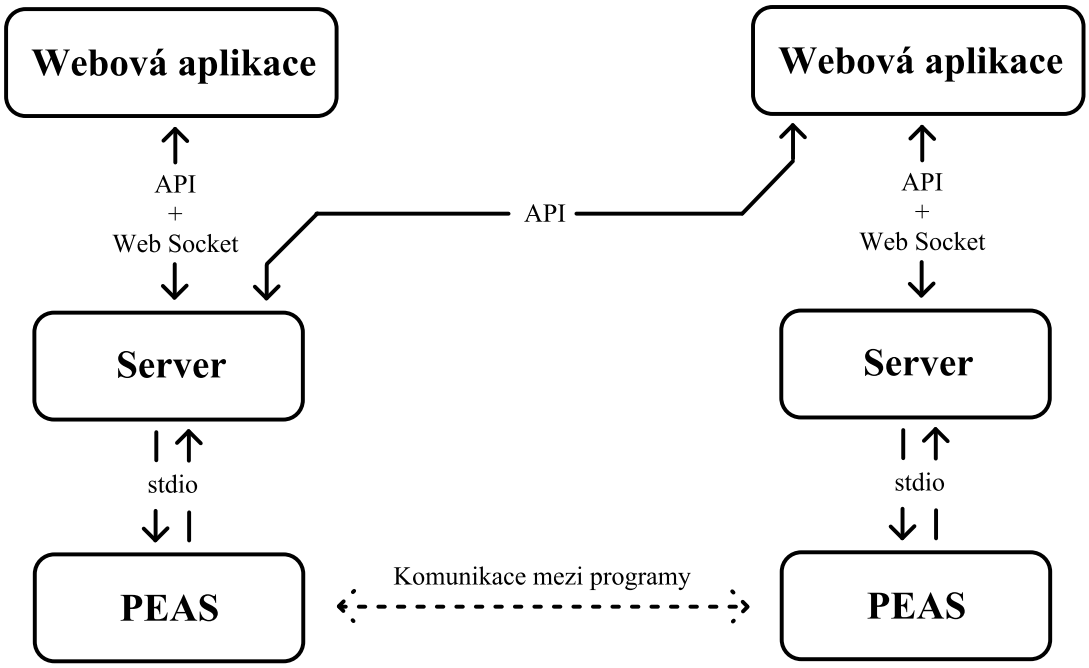
Největší překážkou v implementaci API bylo zajištění komunikace mezi stranami uživatele (*User*) a vydavatele (*Issuer*). Při žádosti uživatele o přidělení nových atributů, je na straně vydavatele spuštěno interaktivní dotazování na jejich vyplnění. V tu dobu je potřeba, aby zde byly již připraveny vstupy pro automatické vyplnění těchto údajů do konzole. Proto bylo zapotřebí pomocí komunikace mezi webovými aplikacemi zajistit předně vyplnění potřebných dat skrze *Issuer* GUI. Celý proces je popsán v následujících bodech.

1. Vyslat pomocí API žádost od *User* aplikace na *Issuer* server
2. Skrze *Issuer* GUI jsou vyplněna data
3. *Issuer* server informuje *User* aplikaci, že je připraven
4. *User* server zadává příkaz `./peas_ue -i`
5. Do programu PEAS strany *Issuer* jsou automaticky vložena data
6. Uživateli je zobrazena hláška o uspěšné registraci

Web socket – je důležitým rozšířením o komunikaci v téměř reálném čase. Funguje v tandemu s API a doplňuje ji o důležitou signalizaci, pro kterou není sama API vždy vhodná. Hlavní výhodou je ustanovování pouze jedno spojení, kterým jsou následně posílány zprávy. V případě API je pro každý přenos potřeba nového spojení. Příkladem využití *web socket* je kontrola, zda má aplikace spojení s webovým serverem. Jde o úspornější řešení, než případ, kdy by se pomocí API musela aplikace opakovaně dotazovat.

Dalším využitím je například automatické oznamování o stavu spuštěných programů PEAS. Uživatel je tak téměř v reálném čase informován o tom, zda je program PEAS spuštěn, nebo zda byl ukončen a s jakým kódem. Největším rozdílem oproti API je ten, že si o data nežádá webová aplikace, ale jsou ji doručována v závislosti na jejich dostupnosti. Webová aplikace pak pouze roztřídí příchozí zprávy dle toho, jaká přenášejí data a následně zprostředkuje jejich obsah uživateli.

Web socket je spojena s ostatní komunikací na portu 80. Při ustanovování spojení, je požadavek zachycen spolu s ostatními na tomto portu. Obsahuje však příkaz *upgrade conection*, který server rozpozná a povýší ustanovované spojení na *web socket*.



Obr. 6.10: Schéma komunikace mezi aplikacemi.

6.3 Logo

Jako prvek, který by měl jednoznačně identifikovat celý produkt, bylo navrženo a zpracováno logo. Vychází z komerčního názvu celého systému, tedy PEAS. Jedná se o akronym ze slov *Privacy-Enhancing Authentication System*. V překladu z anglického jazyka tato zkratka znamená hrášek. Byl tedy vznešen požadavek tuto skutečnost nějakým způsobem promítnout i do samotné podoby loga.

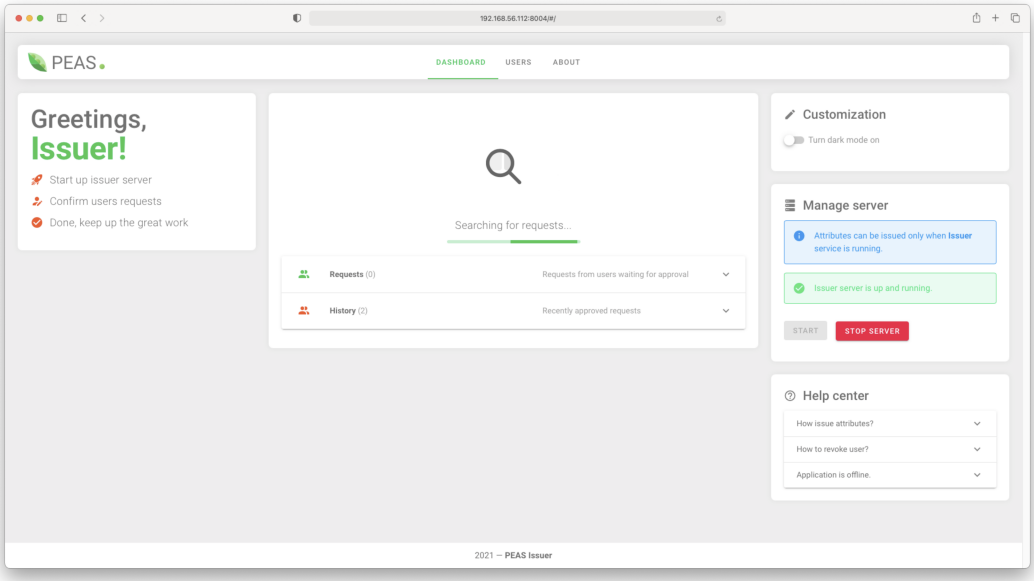
Výsledkem je finální grafika na obrázku 6.11, která má reprezentovat lusk. Doplňuje ji zmiňovaný akronym PEAS ukončený tečkou v podobě kuličky hrášku. Jednotlivé hrášky pak mají být metaforou pro atributy autentizačního systému uložené uvnitř pověření, tedy lusku.

Obr. 6.11: Velká verze loga, bez doplňujících popisků.

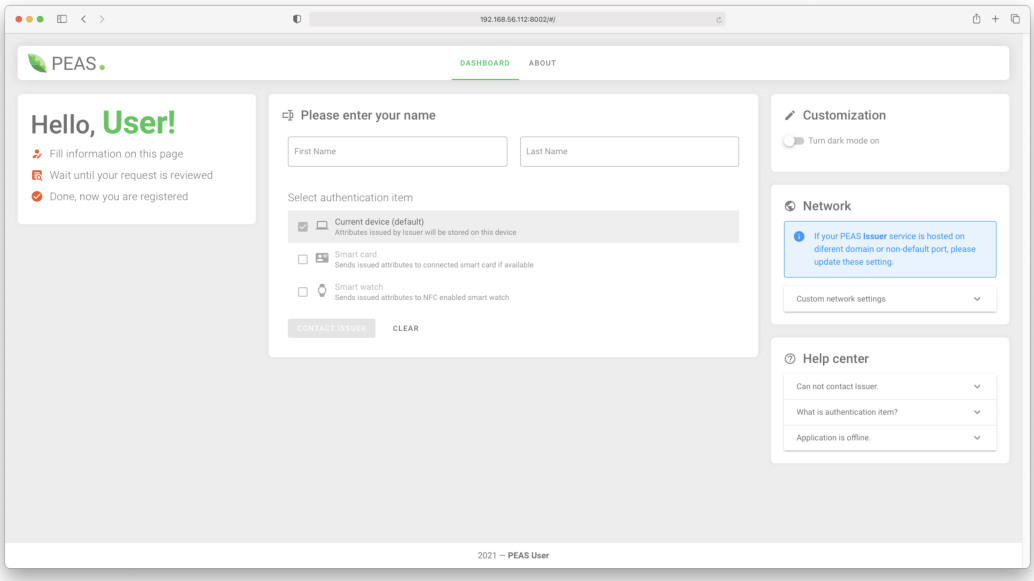
Společně s tímto logem bylo vypracováno několik jeho dalších variant. Liší se rozměry a doplňkovým označením. To je dělí do dvou skupin, pro použití na webu a v mobilní aplikaci. Všechny tyto soubory doplňuje stručný grafický manuál pro jejich správné použití, definující základní barvy a typografii. Lze jej nalézt, včetně balíčku s grafickými soubory loga, v příloze. B

6.4 Výsledná podoba

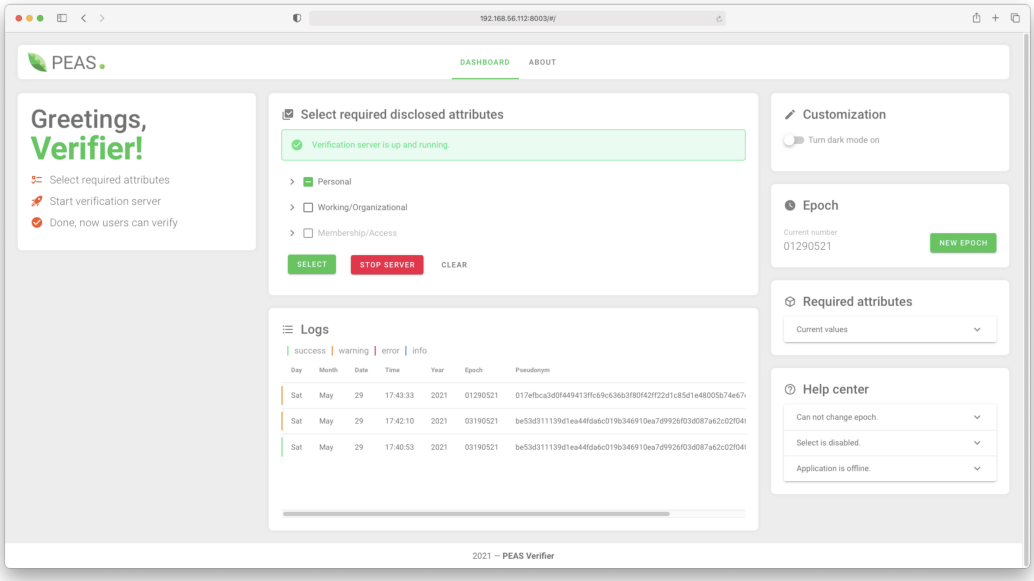
Podkapitola pouze pomocí obrázků prezentuje výsledné podoby webových aplikací. Obrázek 6.12 zobrazuje aplikaci ověřovatele (*Issuer*), obrázek 6.13 rozhraní uživatele (*User*), obrázek 6.14 stránku ověřovatele (*Verifier*) a poslední obrázek 6.15 demonstruje podobu GUI revokační autority (*Revocation Authority*).



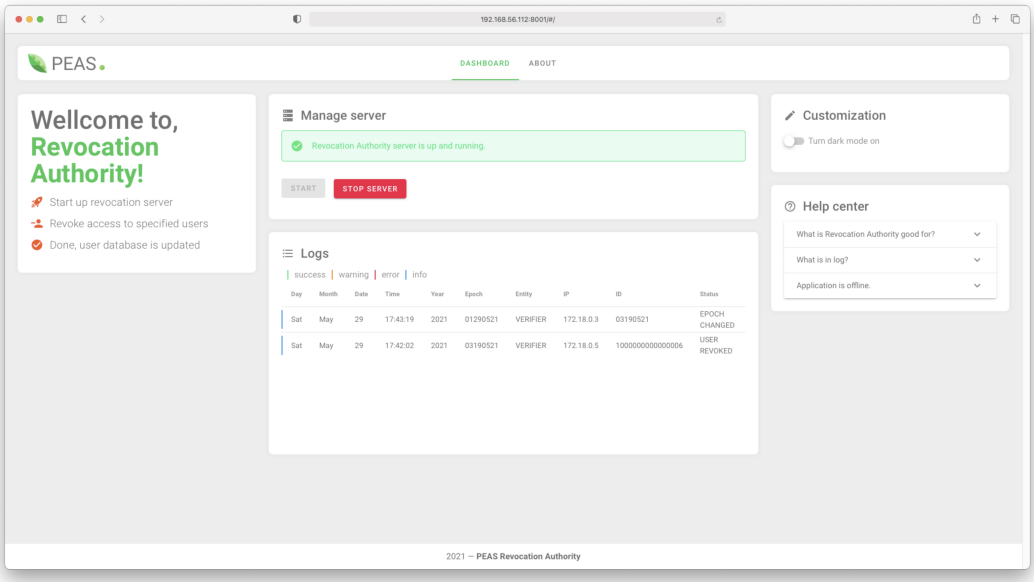
Obr. 6.12: Výsledná podoba aplikace vydavatele.



Obr. 6.13: Výsledná podoba aplikace uživatele.



Obr. 6.14: Výsledná podoba aplikace ověřovatele.



Obr. 6.15: Výsledná podoba aplikace revokační autority.

7 Testování implementace a automatizace nasazení

Kapitola v první části obsahuje metriky vyrobených finálních aplikací, které jsou součástí přílohy. V její druhé části jsou uvedeny podrobnosti k vylepšené automatizaci nasazení celého systému.

7.1 Měření

Záměrem této podkapitoly je naměřit nejdůležitější parametry implementovaných aplikací. Za ty jsou považovány rychlosti načítání grafických rozhraní a ověření uživatele systému.

7.1.1 Statistiky načítání aplikací

Při zvolení webového grafického rozhraní je důležité nejen, aby bylo responsivní v průběhu používání, ale také aby se velmi rychle načítalo. To lze do jisté míry ovlivňovat pokud se jedná o webové rozhraní, které je hostováno a zároveň využíváno na stejném zařízení. Uživatelské rozhraní vytvořeno v této práci je však stavěno i pro použití ze vzdáleného klienta, kdy kvalita připojení mezi ním a serverem není v moci vývojáře.

Proto jedinou cestou jak se pokusit zajistit co možná nejkratší dobu načítání je zmenšit veškeré přenášené asety na minimum. O to se stará především samotný framework, který před nasazením do produkce provede *build* celého projektu a optimalizaci a minifikaci[1] kódu. Zároveň s tím dokáže na základě velikosti výsledné aplikace generovat i statistiky rychlosti načítání v závislosti na různých typech síťového připojení.

Tab. 7.1: Statistiky rychlostí načítání aplikací v závisloti na připojení.

Webová aplikace	Mobile Edge [s]	3G [s]	LTE [s]	Průměr [s]
Vydavatel	306,06	45,93	8,31	10,49
Uživatel	242,63	36,42	4,91	8,32
Ověřovatel	235,93	35,41	4,77	8,09
Revokační Autorita	230,73	34,63	4,67	7,91

[1]Minifikace je proces, při kterém jsou odebrány veškeré nepotřebné znaky (např. mezery, komentáře a formátování) ze zdrojového kódu, bez změny jeho funkcionality. Výsledkem je kompaktní kód o menší velikosti, optimalizovaný pro rychlý přenos po síti.

Konkrétní uvažované parametry jednotlivých spojení:

- **Mobile Edge** – 0,24 Mbps + 840 ms RTT[2]
- **3G** – 1,6 Mbps + 150 ms RTT
- **LTE** – 12 Mbps + 70 ms RTT
- **Průměr** – 7 Mbps + 30 ms RTT

7.1.2 Rychlosti ověření uživatele skrze grafické rozhraní

Systém PEAS disponuje několika funkcemi, ale ne všechny budou v praxi používány se stejnou četností. Například procesem registrace si uživatel systému projde pouze jednou a proto je u takové operace přípustná vyšší časová náročnost. Nejčastějším úkonem, který bude uživatel tohoto systému v praxi provádět, je ověření.

Proto bylo provedeno měření rychlosti této funkce. Byl zaznamenáván čas v milisekundách, který dělil okamžik stisknutí tlačítka od získání odpovědi ze serveru zpátky v uživatelské aplikaci. Naměřeno bylo celkem šedesát pokusů o ověření z toho třicet s výsledkem *access allowed* a třicet s výsledkem *access denied*. V tabulce 7.1.2 lze vidět průměrné časy z naměřených hodnot.

Tab. 7.2: Rychlosti ověření uživatele skrze grafické rozhraní.

Výsledek ověření	Průměrný čas [ms]
Allowed	30,37
Denied	23,87
Společně	27,12

Z výsledků měření je vidět, že hodnoty zpoždění u procesu ověření jsou pro uživatele zcela zanedbatelné. Kompletní tabulka naměřených dat je umístěna v elektronické příloze práce.

7.2 Automatizace

Automatizace nasazení má za úkol usnadnit instalaci a správu celého demonstračního systému. Jde o pokračování, které navazuje na využití Dockeru. Díky němu a souborům *Dockerfile* byl již v minulosti automatizovaný proces kompilace programu PEAS, kryptografické knihovny RKVAC a dalších závislostí. Výsledkem je kompletní Docker *image* obsahující program, webový server s aplikací, *runtime* Node.js a vše nezbytné. Automatizace navržená v této práci přidává automatizované vytváření těchto Docker *image*, Docker kontejnerů, jejich sítě a podobně.

[2]Round Trip Time udává odezvu spojení v milisekundách.

První částí je skript `install.sh`, který obsahuje jednoduchý příkaz na instalaci programu *ansible* pomocí správce balíčků *apt*. Tím je zaručeno, že i když zbylé instrukce jsou psány pro program *ansible*, je možno instalaci spustit i na zařízeních bez této platformy. Spustitelný skript je součástí elektronické přílohy práce.

Ansible je produktem společnosti RedHat. Ten umožnuje vytvářet a automatizovat různé scénáře v podobě *ansible playbooks*. Tyto *playbooky* jsou soubory s instrukcemi psanými ve formátu YAML (*YAML Ain't Markup Language*). Specifickou výhodou *ansible*, je možnost vykonávat předdefinované činnosti na libovolném vzdáleném stroji, nebo skupině strojů bez toho, aniž by tyto servery musely mít nainstalovaného jakéhokoliv agenta. V případě této automatizace je ale definován pouze jeden host a to *localhost*.

Další výhodou použití *ansible* je kontrola prostředí před každým vykonávaným krokem. Pokud je již dosaženo cílového stavu, který by měl nastat po provedení definovaných operací v *ansible playbook*, je krok přeskočen. Tím je značně zvýšena efektivita a rychlost provádění těchto instrukcí.

Vytvořený *playbook* je dělený do několika hlavních oddílů.

1. Instalace závislostí pro přidávání repozitářů
2. Přidání repozitáře Docker
3. Instalace platformy Docker
4. Stažení Docker *image* PEAS platformy
5. Vytvoření Docker sítě
6. Automatické vytvoření Docker *image* pro jednotlivé strany systému
7. Vytvoření a spuštění Docker kontejnerů

Díky těmto krokům, je možné skript použít pro nasazení systému i na zařízení bez jakýchkoliv závislostí. Zmiňovaná vlastnost *ansible*, která vykonává pouze potřebné kroky, jej dělá ideálním i pro instalaci aktualizací a nových konfigurací. Pokud je programem detekovaná nová verze kódu v instalační složce, jsou vytvořeny nové Docker *image* obsahující aktualizovaný kód, poté jsou zastaveny současné kontejnery a místo nich spuštěny nové z aktualizovaných Docker *image*. Jediným úkolem uživatele je tedy stáhnout tento nový kód. Zde je místo pro další vylepšení v podobě automatizování i tohoto kroku.

Poslední funkcí *ansible* je možnost použít vyhrazené YAML soubory jako konfigurační. Toho bylo využito a v adresáři `/vars/` je připraven soubor `containers.yaml` obsahující základní konfiguraci instalace. Je zde možné zvolit platformový *image*, s jakým příkazem budou kontejnery automaticky spouštěny apod. Lze upravit i samotnou konfiguraci kontejnerů od jejich jména po IP adresy a porty.

Závěr

V teoretické části práce byl nejprve uveden pojem webové aplikace 1. Dále byla v kapitole 2 věnována pozornost porovnávání a hledání vhodného programovacího jazyka a frameworku pro tvoru požadovaných webových aplikací. Po nalezení a zhodnocení těch nejpopulárnějších variant byla na základě porovnání v podkapitole 2.4 vybrána verze využívající jazyk JavaScript a framework Vue.js.

Navazující kapitola 3 byla zpracována jako teoretická příprava a analýza možností jak provázat vytvářené webové aplikace s existujícím programem PEAS a knihovnou RKVAC. Výsledkem bylo nalezení dvou cest. První obnášela vytvoření vlastního API, za jehož pomoci by mohla aplikace vzdáleně komunikovat se serverem. Druhá se zaměřovala na integraci kódu jazyka C, ve kterém je psán celý program PEAS, přímo do webových aplikací.

Z důvodu existence druhé varianty byla poté v sekci 3.2.1 blíže zkoumána také technologie *WebAssembly* a kompilátor Emscripten. V rámci předcházející semestrální práce byla i prakticky tato varianta integrace úspěšně odzkoušena, ale nakonec z důvodů uvedených v závěru sekce 6.1.1, nebyla ve finální implementaci použita.

Poslední teoretická kapitola 5 přiblížila základní pravidla, jimiž by se měl řídit vývoj nejen webových aplikací, ale veškerých uživatelských rozhraní. V několika obecných pravidlech, která byla uplatněna i při návrhu GUI tvořeného v této práci, přibližuje zásady, které by měly vést ke kvalitnímu UX.

V praktické části, která je popsána především v kapitole 6, byla navržena a úspěšně implementována čtyři uživatelská rozhraní pro autentizační systém.

Nejprve bylo rozhodnuto o finální architektuře, která bude využita na tvorbu webových aplikací. Bylo vybíráno ze zmiňovaných dvou variant připravených v teoretické části práce. Zvolena byla varianta využívající API pro komunikaci mezi programem PEAS a GUI. Poté bylo již přistoupeno k samotnému vývoji a implementaci aplikací za pomocí dříve připravených nástrojů, frameworků a technologií.

Při tvorbě GUI byl řešen nespočet problémů, kdy se představy nabyté v teoretické části práce lišily, od chování zmiňovaných nástrojů v praxi. Největší překážkou bylo vytvoření efektivní a spolehlivé komunikace mezi webovým serverem a programem PEAS. Tato problematika je zachycena v oddílu 6.2.2. Díky využití runtime prostředí Node.js, které nabízí vestavěné funkce na komunikaci s externími programy, byla nakonec i tato část implementace v pořádku dokončena.

Část řešení byla věnována i vytvoření ucelené grafické podoby a reprezentace celého systému. Toho bylo docíleno navržením a vytvořením loga, jehož ukázku je možné najít v podkapitole 6.3. Zároveň s ním byl i vypracován stručný grafický manuál, ve kterém jsou definovány použité barvy a typografie. Ten je včetně dalších několika variant loga dostupný v elektronické příloze této práce.

Na závěr praktické části této práce byla věnována pozornost dalšímu vylepšení automatizace, při nasazování autentizačního systému do provozu. Celý systém PEAS byl již předem připraven pro použití s kontejnery Docker. Podkapitola 7.2 se proto zaměřila na automatizaci vytváření a správy těchto kontejnerů.

Výstupem celé práce je tedy funkční webové rozhraní pro každou ze stran autentizačního systému PEAS. Podařilo se implementovat veškeré požadované funkce. V současné chvíli lze ovládat a komunikovat s programy PEAS pomocí GUI minimálně stejně tak dobře, jako pomocí vestavěného CLI.

Nejedná se však o podobu, která by neměla co zlepšovat. Cílem při implementaci bylo poskytnout především solidní základ, na kterém by mohl stavět případný další rozvoj grafického rozhraní pro tento systém a tento cíl byl splněn.

Literatura

[1] JOBE, William. *Native Apps vs. Mobile Web Apps* [online]. Stockholm, Sweden, 2013 [cit. 2020-10-29]. Dostupné z: https://onlinejour.journals.publicknowledgeproject.org/index.php/i-jim/article/view/3226/2840. Paper. Stockholm University.

[2] GHIMIRE, Devndra. *Comparative study on Python web frameworks: Flask and Django.* [online]. Metropolia University of Applied Sciences, 2020 [cit. 2020-10-29]. Dostupné z: https://www.theseus.fi/bitstream/handle/10024/339796/Ghimire_Devndra.pdf?sequence=2&isAllowed=y. Bachelor of Engineering. Metropolia University of Applied Sciences.

[3] OSMANI, Addy. *Getting started with Progressive Web Apps* [online]. 2015 [cit. 2020-11-04]. Dostupné z: https://addyosmani.com/blog/getting-started-with-progressive-web-apps/.

[4] BIØRN-HANSEN, Andreas, Tim A. MAJCHRZAK a Tor-Morten GRØNLI. Progressive Web Apps: The Possible Web-native Unifier for Mobile Development. *Proceedings of the 13th International Conference on Web Information Systems and Technologies* [online]. SCITEPRESS - Science and Technology Publications, 2017, 344-351 [cit. 2020-11-04]. ISBN 978-989-758-246-2. Dostupné z: doi:10.5220/0006353703440351

[5] CHAPMAN, Stephen. Introduction to JavaScript. *ThoughtCo.* [online]. Aktualizováno 19. 1. 2019 [cit. 2020-11-28]. Dostupné z: https://www.thoughtco.com/what-is-javascript-2037921

[6] RAUSCHSMAYER, Axel. *Speaking JavaScript* [online]. 30. 1. 2015, Druhé vydání. United States of Amerika: O'Reilly Media, 2014 [cit. 2020-11-28]. ISBN 978-1-449-36503-5. Dostupné z: https://books.google.cz/books?hl=cs&lr=&id=tBbsAgAAQBAJ&oi=fnd&pg=PR4&dq=javascript+language&ots=HGJ-8GhbxJ&sig=hblN8uY4FIUMSC4w-0ZGnMah1xQ&redir_esc=y#v=onepage&q=javascript%20language&f=false

[7] TEIXEIRA, Pedro Dennis. *Professional Node.js: building JavaScript-based scalable software* [online]. Indianapolis: Wiley, c2013 [cit. 2020-11-28]. Wrox programmer to programmer. ISBN 978-111-8185-469. Dostupné z: https://books.google.cz/books?hl=cs&lr=&id=ZH6bpbcrlvYC&oi=fnd&pg=PR27&dq=node+js&ots=mPzucCpoPd&sig=V2x4_nEkZqp0JQSvF98g5QnQ6Og&redir_esc=y#v=onepage&q=node%20js&f=false

[8] Node Js: Non-blocking or asynchronous | Blocking or synchronous. *CronJ* [online]. 2018 [cit. 2020-11-29]. Dostupné z: https://www.cronj.com/blog/node-js-non-blocking-asynchronous-blocking-synchronous/

[9] WOHLGETHAN, Eric. *Supporting Web Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue.js* [online]. Hamburg, 2018 [cit. 2020-11-29]. Dostupné z: https://reposit.haw-hamburg.de/bitstream/20.500.12738/8417/1/BA_Wohlgethan_2176410.pdf. Bakalářská práce. Hamburg University of Applied Sciences.

[10] DRAYTON, Peter, Ben ALBAHARI a Ted NEWARD. *C# in a nutshell: [a desktop quick reference]* [online]. 2nd ed. Sebastopol: O'Reilly, 2003 [cit. 2020-11-11]. ISBN 05-960-0526-1. Dostupné z: https://books.google.cz/books?hl=cs&lr=&id=bG_Aqb6iOUYC&oi=fnd&pg=PR13&dq=c%23+in&ots=PFN5uHeHt9&sig=hKFHWB-qhsV7o5GyXD25YIsbQQk&redir_esc=y#v=onepage&q=c%23%20in&f=false

[11] COCHRAN, Matthew. Introduction to Model View Control (MVC) Pattern using C#. *C#Corner* [online]. Aktualizováno 7. 5. 2019 [cit. 2020-11-29]. Dostupné z: https://www.c-sharpcorner.com/article/introduction-to-model-view-control-mvc-pattern-using-C-Sharp/

[12] WYATT, Matt. What is an API? A Digestible Definition with API Examples for Ecommerce Owners. *BigCommerce* [online]. [cit. 2020-11-25]. Dostupné z: https://www.bigcommerce.com/blog/what-is-an-api/#so-what-is-json-and-why-is-it-used

[13] FERNANDEZ, Tomas. What is WebAssembly? *Stackpath* [online]. 5. 9. 2019 [cit. 2020-11-28]. Dostupné z: https://blog.stackpath.com/webassembly/

[14] PROTZENKO, Jonathan, Benjamin BEURDOUCHE, Denis MERIGOUX a Karthikeyan BHARGAVAN. Formally Verified Cryptographic Web Applications in WebAssembly. *2019 IEEE Symposium on Security and Privacy (SP)* [online]. IEEE, 2019, 2019, , 1256-1257 [cit. 2020-11-28]. ISBN 978-1-5386-6660-9. Dostupné z: doi:10.1109/SP.2019.00064

[15] MAINO, Elia. Webassembly: Calling C functions from Javascript with emscripten. *Medium* [online]. 27. 7. 2017 [cit. 2020-11-28]. Dostupné z: https://medium.com/@eliamaino/calling-c-functions-from-javascript-with-emscripten-first-part-e99fb6eedb22

[16] ANDERSON, Jim. Python Bindings: Calling C or C++ From Python. *RealPython* [online]. 2. 3. 2020 [cit. 2020-11-28]. Dostupné z: `https://realpython.com/python-bindings-overview/`

[17] *Data marshalling* [online]. [cit. 2020-12-06]. Dostupné z: `https://www.webopedia.com/TERM/D/data_marshalling.html`

[18] KEREN, Guy. Building And Using Static And Shared "C"Libraries. *docencia.ac.upc.edu* [online]. 1998 [cit. 2020-11-28]. Dostupné z: `https://docencia.ac.upc.edu/FIB/USO/Bibliografia/unix-c-libraries.html`

[19] FISHER, Tim. What Is a DLL File?: DLL Files: What they are & why they're important. *LifeWire* [online]. 13. 10. 2020 [cit. 2020-11-29]. Dostupné z: `https://www.lifewire.com/what-is-a-dll-file-2625852`

[20] *DllImport Attribute* [online]. 2011 [cit. 2020-11-29]. Dostupné z: `https://www.techopedia.com/definition/25611/dllimport-attribute`

[21] HLINKA, Jan. *Kryptografie na výkonově omezených zařízeních*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce Doc. Ing. Jan Hajný, CSc.

[22] HAJNÝ, J.; DZURENDA, P.; CASANOVA MARQUÉS, R.; MALINA, L. Privacy ABCs: Now Ready for Your Wallets!. *In Proceedings of The 19th International Conference on Pervasive Computing and Communications (IEEE PerCom 2021)*, 2021. s. 686-691 [cit. 2021-05-11]. ISBN: 978-0-7381-4348-4.

[23] HAJNÝ, J.; DZURENDA, P.; CAMENISCH, J.; DRIJVERS, M. Fast Keyed-Verification Anonymous Credentials on Standard Smart Cards. In ICT Systems Security and Privacy Protection. *Springer Nature Switzerland*, 2019. s. 286-298 [cit. 2021-05-11]. ISBN: 978-3-030-22312-0.

[24] CAMENISCH, J.; DRIJVERS, M.; HAJNÝ, J. Scalable Revocation Scheme for Anonymous Credentials Based on n-times Unlinkable Proofs. *In WPES '16 Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society*. NY, USA: ACM New York, 2016. s. 123-133 [cit. 2021-05-11]. ISBN: 978-1-4503-4569-9.

[25] HAJNÝ, J.; MALINA, L.; DZURENDA, P. Privacy-PAC: Privacy-Enhanced Physical Access Control. *In WPES 2014 Proceedings*. USA: 2014. s. 1-4 [cit. 2021-05-11]. ISBN: 978-1-4503-3148- 7.

[26] BABICH, Nick. *The 4 Golden Rules of UI Design* [online]. 7. 10. 2019 [cit. 2020-11-23]. Dostupné z: https://xd.adobe.com/ideas/process/ui-design/4-golden-rules-ui-design/

[27] NIELSEN, Jakob. *10 Usability Heuristics for User Interface Design* [online]. 24. 4. 1994; Aktualizováno 15. 1. 2020n. l. [cit. 2020-11-23]. Dostupné z: https://www.nngroup.com/articles/ten-usability-heuristics/

[28] WONG, Euphemia. *User Interface Design Guidelines: 10 Rules of Thumb* [online]. [cit. 2021-5-11]. Dostupné z: https://www.interaction-design.org/literature/article/user-interface-design-guidelines-10-rules-of-thumb

[29] NIELSEN, Jakob. *Enhancing the Explanatory Power of Usability Heuristics* [online]. 24-28. 4. 1994, , 125-158 [cit. 2021-5-11]. Dostupné z: https://static.aminer.org/pdf/PDF/000/089/679/enhancing_the_explanatory_power_of_usability_heuristics.pdf

[30] *2020 Developer Survey* [online]. 2020 [cit. 2020-12-06]. Dostupné z: https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages

[31] Material Design: Introduction. *Material.io* [online]. [cit. 2021-5-2]. Dostupné z: https://material.io/design/introduction

Seznam symbolů, veličin a zkratek

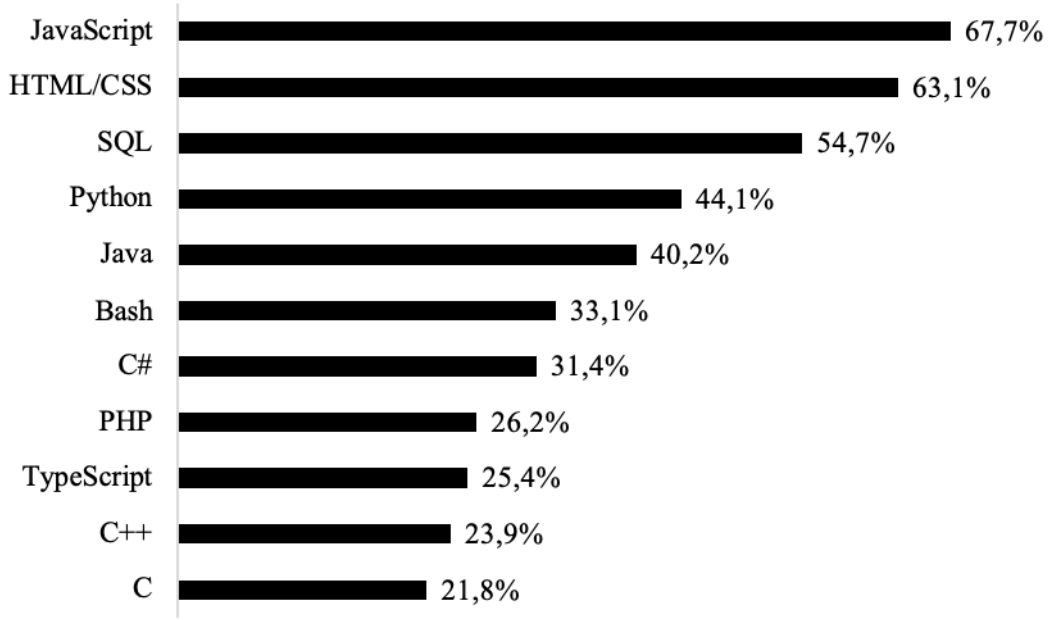
API	rozhraní pro programování aplikací – Application Programming Interface
CLI	rozhraní příkazové řádky – Command Line Interface
CLR	Common Language Runtime
CPU	Centrální procesorová jednotka – Central Processing Unit
CSS	kaskádové styly – Cascading Style Sheets
DLL	sdílená knihovna – Dynamic Link Library
eIDAS	Electronic Identification and Trust Services
FCL	Framework Class Library
GDPR	General Data Protection Regulation
GUI	grafické rozhraní – Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JS	JavaScript
JSON	JavaScript Object Notation
JSX	syntaktické rozšíření JavaScriptu
MVC	Model-View-Controller
PEAS	Privacy-Enhancing Authentication System
PHP	skriptovací programovací jazyk
PWA	Progressive Web Applications
RKVAC	kryptografická knihovna programu PEAS
TS	TypeScript
URL	Uniform Resource Locator

UX	User Experience
W3C	World Wide Web Consortium
YAML	rekurzivní akronym – YAML Ain't Markup Language

Seznam příloh

A Data z průzkumu StackOverflow

Graf A.1 znázorňuje nejpoužívanější jazyky vývojářskou komunnitou StackOverflow. Celkem jej tvoří údaje od 57 378 respondentů, kteří vybírali všechny jazyky jenž používají.



Obr. A.1: Průzkum na stránce StackOverflow. [30]

B Obsah přiloženého CD

V přiloženém médiu jsou dostupné zdrojové kódy webových aplikací, serverů a automatizačních skriptů. Dále obsahuje adresář s připraveným prostředím pro instalaci a běh celého dema systému PEAS včetně webového uživatelského rozhraní. Součástí je taktéž instalační a uživatelský manuál. Nachází se zde i adresář obsahující všechny varianty loga, grafický manuál a další ilustrace vytvořené v rámci této práce. Níže jsou uvedeny verze softwaru použitého pro práci s těmito soubory.

Upozornění – Z důvodu překročení limitu velikosti přílohy, která v IS VUT činí 15MB, byla elektronická příloha práce umístěna na externí cloudové úložiště. Lze jí stáhnout z veřejně dostupné adresy: `https://drive.google.com/file/d/1tC1-nQgdhWkfPKY6kI8nGXLkOnklln2L/view?usp=sharing`

Verze použitého software

- **Visual Studio Code** – 1.51.1
- **Node.js** – 12.19.0
- **Vue CLI** – 4.5.8
- **Vuetify** – 2.4.11
- **Emscripten** – 2.0.9

Obsah

```
/ ..... kořenový adresář přiloženého CD
├── PEAS-Docker ..... adresář obsahující prostředí pro tvorbu Docker kontejnerů
├── PEAS-Servers ..... adresář obsahující projekty webových serverů
│   ├── issuer
│   ├── revocation-authority
│   ├── user
│   └── verifier
├── PEAS-WebApps ..... adresář obsahující projekty webových aplikací
│   ├── issuer
│   ├── revocation-authority
│   ├── user
│   └── verifier
├── UI-Assets ..... adresář obsahující grafické assety a loga včetně manuálu
│   ├── Animations
│   ├── Graphic Design Manual
│   ├── Illustrations
│   └── Logo
├── Manual.pdf ..... uživatelský a instalační manuál dema
├── Mereni.xlsx ..... naměřená data
└── README.md ..... soubor s obecným popisem projektu
```